The Pragmatic Programmers



图灵程序设计丛书

卓越程序员密码

The Developer's Code

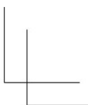
What Real Programmers Do

- 避免常犯的错误
- 养成优良的习惯
- 破解密码，你也可以卓越起来

[美] Ka Wai Cheung 著
劳佳 译



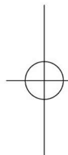
人民邮电出版社
POSTS & TELECOM PRESS

本书讲的不是你写的代码，而是你赖以生存的密码。

软件开发行业的从业人员成千上万，你如何保证自己出色地完成工作？这本书里没有浮夸的老生常谈，作者汇集十余年来的软件开发经验，从各个角度审视这一行业，探讨了保持健康工作状态需要怎么去做。

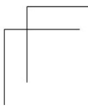
如何在最漫长的项目中保持效率，如何建立一个适合自己而不是牵绊自己的工作流程，如何面对目标和你不一致的客户……日常工作中遇到的许多问题，都出现在作者的笔端。有些问题如果处理得不好，哪怕是最有经验、最有干劲的程序员也可能被击垮。但有了正确的手段，你就可以克服这些难题，成为你梦想中的专业程序员。



在这五十多篇智慧小文中，你还会学到：

- 为什么软件行业中针对流程和开发职务的很多传统方法都是错的，以及如何发现这些错误；
- 为什么你必须对消闲项目和没有期限的项目说不；
- 如何把代码生成融入你的开发流程，以及它有什么你想不到的好处；
- 客户和最终用户不同意你选择的方法时怎么办；
- 如何言传身教，将知识传授给下一代程序员。

如果你准备长期从事软件开发行业，相信你会不断地反复阅读这本书的。

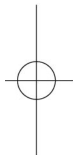







Ka Wai Cheung

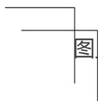
中文名张家为，是程序员、设计师，还是芝加哥We Are Mammoth（我们是猛犸）公司的联合创始人。为各行各业的客户制作应用软件，也制作自有的一些基于网页的软件。





劳 佳

上海交通大学电子工程系硕士，现在SAP美国任高级软件支持顾问。业余爱好语言、数学、设计，近年合作翻译出版了《周末读完英国史》、《加州大学伯克利分校人文建筑之旅》等书。



TURING 图灵程序设计丛书

卓越程序员密码

The Developer's Code

What Real Programmers Do

[美] Ka Wai Cheung 著

劳佳 译



人民邮电出版社

北京

图书在版编目（C I P）数据

卓越程序员密码 / (美) 张家为 (Cheung,K.W.) 著;
劳佳译. -- 北京 : 人民邮电出版社, 2012.11
(图灵程序设计丛书)
书名原文: The Developer's Code:What Real
Programmers Do
ISBN 978-7-115-29508-8

Ⅰ. ①卓… Ⅱ. ①张… ②劳… Ⅲ. ①程序设计－工
程技术人员－工作方法 Ⅳ. ①TP311.1

中国版本图书馆CIP数据核字(2012)第232112号

内 容 提 要

本书集合了作者在软件行业里总结的第一手教训、体会和走过的弯路。话题涉及程序员生活的方方面面，例如，如何保持开发动力，如何提高生产力，如何摆脱软件的复杂性，如何与客户打交道，如何教导编程新手，何时进行自主开发，程序员的自豪感等。每个话题独立成篇、言简意赅，引人思考。

本书不仅适合编程老手阅读，也适合编程菜鸟学习，还适合想了解软件这个行业的人士阅读。

图灵程序设计丛书

卓越程序员密码

◆ 著　　[美] Ka Wai Cheung
译　　劳　佳
责任编辑　卢秀丽
执行编辑　岳新欣

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
邮编　100061　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：880×1230　1/32
印张：5.375
字数：138千字　　2012年11月第1版
印数：1－4 000册　　2012年11月北京第1次印刷

著作权合同登记号　图字：01-2012-6909号

ISBN 978-7-115-29508-8

定价：29.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

版权声明

本书赞誉

“这是 Pragmatic Programmers 系列中的又一本书——对新手来说是指引，对专家来说是重温，这是关于程序员技艺（和生活）的一本美妙的智慧集。”

——Derek Sivers，CD Baby 和 sivers.org 的创始人

“Ka Wai Cheung 先生为那些寻找自己赖以生存的代码的专业开发人员写了一本书。这本书不是用那些在任何博客中都能找到的传统想法拼凑起来的，而是用强有力且有针对性的方法，讲述专业编程的技艺和现实。

如果你想找一本新瓶装旧酒的编程规则，那就不用看这本书了。但是，如果你正在寻找一种视角，看看软件开发是什么，或者你想要一套由真实经验提炼出的指导方针，那这本书正是你需要的。”

——Bob Walsh，作家、47 Hats 的创始人

“充满‘美味’的经验，每篇的大小也十分‘适口’——在这本书里你可以学到很多。花上些时间从过来人那里学学吧。”

——Adam Hoffman, 高级开发主管

“一本好书，有现代程序员从日新月异的世界中得到的提示、技巧和经验教训。从事开发或与开发人员合作的人士不可不看。”

——Caspar Dunant, Webfish

译　者　序

这本小书名叫《卓越程序员密码》，但大部分内容讲的不是具体的技术。书中的话题涉及程序员生活的方方面面，每个话题独立成篇、言简意赅，读来多有切身之感，引人思考。譬如每日邮件繁忙，电话不断，如何保证工作效率？有些程序员朋友，自身水平很高，为何向别人讲解问题时却效果不佳？如何和不断提要求的客户周旋？作者在行业中浸淫多年，在团队建设、项目管理等方面都有独到的见解。当然，书中更少不了关于软件开发本身的技巧，复杂性管理、重构、代码生成等都是对实际工作很有指导意义的方法。

如果你从事开发，在上班的路上读上一两篇，也许书中的观点就能给一天的工作带来一点灵感。即使你从来不写程序，读读这本书也有助于了解一下这个热门而有趣的行业。要是你的男朋友刚好是个程序员，这本书或许能让你更好地走近他的生活，理解他的种种行为也说不定呢。

原文笔触轻快，语言流畅，读来丝毫没有读技术文献的枯燥感。作为译者，我也希望尽可能带给读者同样轻松愉快的阅读体验。限于水平和精力，译文不妥之处，还望读者指正。图灵岳新欣编辑和武卫东总编为译稿润色付出了辛勤的劳动，在此特别致谢。

最后值得一提的是，本书的翻译和编辑工作均在图灵社区上在线

完成。文本使用 Markdown 语言描述，实时更新，互动编辑，并逐章发布电子版。这种新的出版模式，消除了译者和编辑之间反复发稿的烦琐，也让读者能够随时追上最新的章节。在如今敏捷出版和移动阅读的大潮中，这一尝试显得尤为合宜，十分可喜。

劳佳
2012 年 8 月

中文版序

亲爱的读者：

在编程的世界里，我们会和各种各样的“语言”打交道。虽然我主要的服务器端开发语言是 C#，但我的工作方法却几乎可以完全应用到 Java、PHP、Ruby 或 Python 上。编程语言虽有不同，核心的编程思想、方法和架构却是高度类似的。我们只是用不同的方式来表达而已。

我们的工作方式也具有普遍性——如何保持干劲、提高成效，教学的重要性，如何与客户合作，如何发现合适的流程，勇于摆脱不好的流程。此外，我猜想，我通过编程学到的人生一课，不仅适合在美国工作的我们，也可能适用于在中国工作的你。

作为在美国出生的第一代华裔，我非常荣幸这本书能够被翻译成我父辈的语言。我在美国长大，在西方文化中生活，同时也有一些中国的传统价值观。我对于应该如何在这个行业工作的看法肯定受到了两种文化的影响。

无论你是刚刚投身编程事业，正在创业，还是在寻找一些新鲜的观点，我都希望这本书在现在和未来都能对你有所启发。

张家为

Dear readers,

In the world of programming, we work with “languages” of all different kinds. Though my primary server-side development language is C#, the way I work almost fully translates to the world of Java, PHP, Ruby, or Python. Core programming ideas, methodologies, and architectures are highly similar between these different coding worlds. We just express them in different ways.

The same is true for how we work - How we stay motivated and productive, the importance of teaching, working with clients, discovering the processes that work just-right and being brave enough to shed the ones that don’t. And, I’d imagine that the life lessons I’ve learned through programming translate not just to how we work in the U.S., but how you might work in China.

As a first-generation American born to Chinese parents, I was tremendously humbled to have this book translated to my ancestral language. I grew up in the United States, living in a western culture while growing up with some traditional Chinese values. My opinions on how we ought to work in this industry have certainly been influenced by both cultures.

Whether your just beginning your career in programming, starting your own small business, or seeking some fresh perspectives, I do hope you find this book valuable now and in the future.

Sincerely, Ka Wai Cheung

致　谢

2010年秋，这本书的初稿差不多已经写好了，于是我开始向几家技术出版社投稿。虽然有些反馈还不错，但一直没有人答应出版。多数出版社的意见主要有两条：一是这类书一般不好卖，二是我需要再加上更多的后续内容才值得出版。

Andy Hunt 和 Dave Thomas 看问题的角度与众不同。因此，首先我想感谢 Andy 和 Dave，他们和我一样，相信这本书在我们这个行业会有一席之地。本书能够忝列 Pragmatic Bookshelf 这套出色的丛书之中，我感到不胜荣幸。

像这样一本书迫切需要一位好编辑——能够客观地评价它，并在作者被盘根错节的细节羁绊的时候，能够从高处俯览全局。我要感谢 Brian P. Hogan 在整个过程中的出色工作。无论是从内容还是叙述方式而言，这本书胜过初稿之处都不可以道里计。

我要特别感谢 Andertoons.com 的 Mark Anderson。他富有创意的漫画和机智遍布全书，平添了几分幽默，让本书大为增色。

感谢 Derek Sivers、Bob Walsh、Caspar Dunant、Colin Yates、Juho Vepsäläinen、Steve Cholerton 和 Kim Shrier 抽出时间，热心仔细审读每一个章节。

这本书的灵感大部分来自于我在 We Are Mammoth 的经验，这是

我和 Craig Byrant 在 2006 年创立的 Web 开发工作室。在过去的五年里，我们不断挑战自己，思索我们对待工作的方式，并彼此坦陈我们的意见。感谢我的团队——Craig、Michael、Mustafa、Tom、Sam、Anthony、Jennifer、Grant 和 Lindsay，你们每天都带给我新的经验和教训。

目　　录

第 1 章
引　　言

我被代码逼疯过——两次。

第一次，是我在大学一年级上的一门编程课。这是一门必修课，我就选了。结果这和我小时候看过无数次的电影场景完全是两码事。我并没有敲上几个简单命令，按下回车，然后就看到一个垃圾桶样的机器人和我说“你好”。

课堂上甚至连个垃圾桶机器人都没有，而全是一堆什么指针啊，内存分配啊，对象实例化之类的。我为了弄清楚这都是些什么玩意儿搞得焦头烂额。不过，有一件事是明摆着了——我根本不是编程的料。

我想成为一个艺术家，或者当个数学家。我想要有创造力还要能够精确——就像人们说的，左脑右脑一起上。编程似乎太偏向左半球了，可是我也想不出什么其他的职业道路，同时能够在两个世界施展。我迷茫了。

短短几年后，互联网的大潮彻底改变了编程的景象。编程突然就变得那么实际，那么触手可及，而且还和设计关系紧密。艺术和逻辑几乎被看做同样重要。第一次，我真的觉得能够享受这份工作，可以把我对于创造力和逻辑的热情都倾注到网络应用上。所以，虽然心怀忐忑，我又回来编程了。

讲句老实话，我回来完全是另有原因。在之前的两年里，我学习

了其他很多学科，都似乎有太多太多的未解之谜。是去构思粒子物理里的大统一理论，还是去找最大的素数？毫无疑问，这些任务简直难到无法想象，让人望而生畏。我实在干不了这个。还有，那个讲存在主义的课也没能把事情说清楚。我是个年轻人，我想要的是答案，而不是越来越多的问题。

编程。我曾经唯恐避之不及的那个科目，现在成了我的避风港。不管怎么说，计算机科学还是人搞出来的。所有的事情肯定都有个答案。我期待的职业生涯，应该能让我这种寻求答案的人大展宏图——你能把甘醇的代码变成顾客、同事和客户常在的笑脸。规则都已经定好了，我们只要做就是了。困难无非就是在代码里面而已，我当时是这么想的。

我第二次来做编程，才发现又上当了——显然，实际远不是这么回事。

1.1　谁是 21 世纪的程序员

就像我在接下来的十五年中发现的那样，隐士不适合搞编程。什么有个超级技术狂，躲在幽暗的地下室里，挥汗如雨几个月，然后拿出最终产品，石破天惊，尽享荣华。绝对不是这么回事。

现在，应用程序是主流。即我们为每一位用户做的那种。客户可能了解我们如何工作，也可能完全没什么概念。我们的交付周期有时候是按照白驹过隙的星期来算的，而不是按照月或者年算的。人可能突然一下子就觉得疲惫不堪，而拖延可能是最容易走的那条路。对于我们——当今的开发人员来说，软件生产中所涉及的障碍已经远远超出了开发环境。

我有个好朋友经常和我开玩笑。“你工作的时候到底都干了些

啥？”她知道我是个程序员，但并不是很确定程序员具体都做些什么。她问我问题时的那种刻薄质问的口气，就像《上班一条虫》[①]里的那个办公室顾问 Bob Slydell 一样。

我是这么和她说的：我是一个无认证但是超级有逻辑的心理学家、理疗师、机械工、外交官、商人和教师，我所在的行业，迄今仍然每时每天都有着新的诠释。

这就是我对当今的程序员能够给出的最简洁的定义了。

1.2 吸取第一手教训

我叫张家为。我是个程序员、设计师，还是芝加哥 We Are Mammoth[②]（我们是猛犸）公司的创始合伙人。我们为各行各业的客户制作应用软件，也制作我们自有的一些基于网页的软件。后面会再谈到这一点。

这本书集合了我自己在行业里总结的第一手的教训、体会和走过的弯路。对于编程老手来说，你可能会发现我写的某些轶事自己也经历过。我们可以一起大笑、欢呼或是流泪。对于刚刚开始征程的菜鸟来说，希望这本书能够为你进入行业的头几年助上一臂之力。

过去十五年中，我得到的教训数不胜数。下面仅举几个这本书里谈到的话题。

- 为什么软件行业中许多传统的开发流程和职责分配已经过时了，以及如何找出这些问题。
- 为什么要对消闲项目说“不”，为什么开放的时间表对生产力

---

① 《上班一条虫》（*Office Space*）是 1999 年出品的一部美国电影。——译者注

② 我们的网站是 http://www.wearemammoth.com，我们的博客网址是 http://blog.wearemammoth.com。

至关重要。
- 协作的工作环境如何大幅提高我们的效率，又如何大大降低我们的效率。
- 如何让代码生成成为开发流程中自然而然的一部分，以及它除了让生成代码更快还有什么好处。
- 如何更好地与不能面对面交流的客户打交道，以及如何面对不管软件做了什么新改动都会大发雷霆的客户。
- 为什么大幅加薪和“员工是我们最宝贵的财富”这句古老的口号并不意味着更好的技术工作。
- 如何认识到软件本身已经太过复杂了。
- 如何成为更好的老师，让我们能够把知识在程序员中一代代传下去。

1.3 这本书写的是我们自己

这本书是写给各种各样的程序员的。不过，它和代码本身关系不大。不管你用的是 C#、Ruby、Python、PHP、Java、JavaScript 抑或是 ActionScript，都没什么要紧的。也无论你是在搞数据库，写服务器端代码，还是做界面脚本，都无所谓。这本书讲的是在标识符和对象这个层面之外，专业程序员的方方面面。

这倒也不是说我们就把编程弃之一旁，会有些地方谈到代码。不过，在谈到代码的时候，我们会用一种不那么技术化的全局眼光来看。我们不会一下子给你列出一堆什么最佳实践或者设计模式之类的。这些东西有很多别的书都讲得很好了，我们只会适时提到一些。

这本书讲的是真正的、现代的程序员是怎样让我们的行业欣欣向荣的。我们开始吧。

第 2 章

比　　喻

程序员编程。设计师设计。但这具体是什么意思呢？还没有什么电视真人秀节目或者好莱坞电影展示过我们到底是如何工作的。先别忙着反驳——你可能想到了电影《社交网络》（*The Social Network*），但它对这个行业的阐释并不具有代表性。所以，要是有人问我是干什么的，我常常要打些比方。这个行业充满了比喻，我们用比喻来向行外人描述自己的工作。

大厨用不着琢磨烹饪要怎么比喻，肉汤要是太咸了，你一下就能尝出来了；音乐家也不用这么拐弯抹角地描述歌曲，一个调子听起来太老套，是因为你早先就听过一样的节奏。别人一下就明白了。这些工作都属于不用解释就很清楚的。管道安装工和砌砖工之类的工作更是不言自明了。

然而，编程可不大一样。普通人看不出来什么样的代码优雅，什么样的代码一团糟。还有，我们这个行业非常新。人类做饭、创作音乐、盖房子都有几千年了，可是考古学家还没在岩壁上发现“人类坐在桌前打字”这种图案。

所以，我们必须用比喻作为“元语言”[1]。这不仅让我们能够把编程的独特性讲述给普罗大众，而且还经常是我们解决软件问题时的决策方式。

① 在语言学上，元语言是指用来谈论、观察和分析另一种语言（对象语言）的语言。它可以是自然语言，如学习外语时用于解释外文的本民族语言，也可以是一套语言符号。——译者注

第 1 篇　谨慎使用比喻

这就是危险的地方。要当心！

有时候，比喻和现实之间的界限可能会模糊。比喻可能让我们过于重视那些不重要的东西，而对真正重要的掉以轻心。

如果太投入这些比喻里面，我们可能做不出最好的决定。比喻会像恶魔一样遮住我们的双眼，让我们看不见事实。所做的决定可能在比喻的语境下非常有道理，但抽丝剥茧到了制作应用软件这个实际业务上，我们很容易就被带偏了。我们有时候过于依赖在比喻上看到的好处，却没有注意到现实的情况。

比如，我们常常用传统的建筑行业来比喻建造软件的过程，毕竟很多职务头衔都是从这个比喻中来的。于是我们给自己取了软件架构师、信息架构师、高级程序员、初级程序员以及项目经理之类的名字，很多人还把线框图、说明书、流程图、甘特图、瀑布开发模型之类奉为圭臬。软件开发流程中很大一部分都是跟在其他行业屁股后面搞出来的。

虽然这些概念在其他行业中很重要，对我们的行业也有那么一些好处，但它们也很容易束住我们的手脚。如果停下来想想为什么一定要用某种特定的方式来解决一个问题，也许就能追根溯源，发现我们对传统建筑行业这个比喻（或其他某个比喻）跟得太紧了。

那么，比喻给我们造成了哪些伤害呢？先来看几个例子，我们把实际建筑行业中的概念硬拉到软件上来，却发现不怎么合适。

第 2 篇　规划完备，然后开工

在传统建筑行业中，规划是至关重要的。有些东西就是必须比另外一些东西先做，这没什么好争辩的。打好壁骨框架才能铺上下水管道，铺好了管道才能装墙面，墙面装上了才能粉刷。要是建个摩天大楼，撤销、剪切或者复原显然是行不通的。

软件的撤销就是 CTRL+Z，软件的剪切就是 CTRL+X，软件的复原就是源码控制里面的代码回滚。



“这儿可没有 Ctrl-X，剪刀就是干这个用的。”

建筑上没法享受这些简洁而强劲的按键，所以需要非常详尽的说明书。日进斗金的房地产生意和灾难性的头条新闻之间就只有一步之遥。

假设我们是传统的建筑师，拥有软件开发的各种快捷方式，那么这世界真是太美好了。原材料取之不尽。我们可以几个星期就搞定一个实际大小的建筑模型，还可以对悬索桥反复做压力测试。要是桥断了，谁在乎呢？几分钟就可以复制出十座新桥来！

当然了，这些不过是白日做梦罢了。所以，要是打算建造一栋摩天大楼，把建筑说明写得详细到让人想吐才是最合理的做法。

反过来，这些就是我们这个行业独有的奢华享受了。软件组件又不需要等着本地的工厂发运字母和数字。打字、编译、测试，然后重复就行了。我们可以在实际产品上测试代码，而不用对着产品的某种模型测试。在开发过程中，我们可以看着悬索桥断上千百次，在各种地方断，在各种条件下断，而不用担心浪费材料或者闹出人命。这么做完全是可行的。完成软件之后，同样的程序可以被复制 1000 次，不费吹灰之力。

拉斯韦加斯的 Wynn 大酒店的开发商在 2008 年又建了一个几乎一模一样的酒店，叫做 Encore。他们可没法简单地把前一个作品复制粘贴到旁边的空地上。他们还得从头开始设计、规划，只是为了建一个几乎一模一样的建筑。

在软件还需要用磁盘承载代码的年代，进行完备的规划还比较有意义。然而，基于 Web 的软件完全是另一回事。在写第一行代码之前就做出非常详细的说明书仍然有些好处，但这没能充分利用到这种媒介的优势。每天、每小时或是随便什么时间，我们都可以坐在舒适的 Aeron 座椅里面发布版本，而且基本没什么成本。

好在，作为一个行业，我们已经开始打破这个比喻的樊笼。敏捷开发并不是什么革命性的东西，无非是把我们从一个旧比喻的束缚里面解脱出来，因为旧的做法在今天已经不像在过去那么有价值了。这并不是说，传统的瀑布开发模型已经过时了，它对于比较复杂的大软件项目仍然有其好处。然而不假思索地遵循这个旧比喻也可能遮住我们的双眼，妨碍我们采用一种更适合自己环境的新方法。

“规划、规划、规划”的比喻过分强调要花大量时间计划让所有东西臻于完美，而忽视了可以用好实际写代码的工夫。

第 3 篇　发行不过是第 1 版

传统上，我们把发行日期看做一个关键任务点，软件到这时必须是最终版。没有回头路了。

对于建筑物和大楼结构来说，这是最基本的。对软件而言，这个比喻曾几何时也是有道理的。把软件装在软盘和 CD 上发货的时候，什么东西都得弄对了。有了 bug 就意味着要花费巨大的成本和大量的时间。为了追求完美或是塞进一个新功能，项目就得不断拖延。我在下一章将谈谈这对士气会有什么影响。

今天，基于 Web 的程序不再是“发行”了，而是上传、发布、推送了。软件会随着时间的推移而发展和成熟。

发行之后，第 2、3 版乃至第 20 版可以在几天甚至几小时之后发布。甚至软件的正式版本号这个概念也过时了。这只有在用磁盘发行软件的过去才有意义。

现在，我们不停地整合，不断地反复。和汽车行业不一样，我们用不着大批量召回。如今，遇到关键的 bug 可以立刻打补丁，进行测试，加以部署。它不再是 2.0 版了，现在是 2.0.12931 版，或者，简单来说就是今天当天的版本。还有谁会因为这点 bug 盯着你不放吗？

社会对于这种迭代也渐渐习以为常了。你有没有看到 Facebook 上的新图片库？你有没有看到 Google 新的自动提示功能？Twitter 的新布局呢？没人再去搞一个月的宣传活动来提醒你了。新的变化就这样天天在出现。IMVU[①]是一个流行的 3D 聊天应用，号称有 1 亿多注册用户，它每天要发布 50 次。

① 参见 http://www.imvu.com。

在如今的系统架构下，最初发行不再像过去那样，或是像很多其他行业那样，必须尘埃落定、功德圆满。它无非是整个软件生命周期中几百个（乃至几千个）小发布中的一个而已。这种观点可以减轻发行软件的精神压力。

不幸的是，这种观念很容易就被滥用了。不要把这种概念转变当成懒惰或者是无限期拖延的借口。应用软件的首次发行版应当是在别人拿到之前就非常非常好了。大的方面都得是对的，需要有完善的安全系统。但是小的方面，那些可以后面再慢慢修复的东西，不应成为发布软件的障碍。你可能会很惊讶，你在发行软件时曾经觉得重要的东西突然就不重要了……毕竟现在已经送出去了。

在软件发行的时候仍然应当庆祝一下，把整个团队拉出去吃一顿大餐。但别把所有的感情都倾注在“婚礼”上，后面还得和这个软件厮磨多时呢，还会做些调整，加入一组新功能，把错的东西搞对，等等。

发行无非是软件生命中的一个点而已，不是尘埃落定，也不是功德圆满。

第 4 篇 “象牙塔”架构师的传说

“技术架构师不应该再写代码”这种说法，我从来都不以为然。

在建筑行业，建筑师躲进了象牙塔，他们的生活里只有规划，不会去敲钉子或是焊接缝。让建筑师去做钻洞或是铺混凝土这种体力活根本就是不可行的。建筑设计和建筑施工这两种职业可谓是泾渭分明。

做官越大，编程越少

在我们这个行当，技术架构师这个职位是通过实实在在的开发得来的——就是建造应用程序的“体力活”。但是，在大多数公司里，当职位逐级攀上软件开发的阶梯时，代码反倒写得少了。我们越来越沉浸在规划里，而不是去第一线发现问题。我们更关心的是总体的图景，而不是代码的细枝末节。在我们这个行业里，一直都有一种理念，就是架构师应该做规划，程序员才应该做开发。

于是造成了一个错误的印象，就是如果你达到了某个水平，亲自编程就不再是最有价值的工作了，只要把脏活累活留给初级程序员就行了！与此同时，这也让低阶层的程序员不再去思考整体目标和项目的走向，我们只是要求他们集中精力做实现。这种架构师–程序员的分工模式让双方对应用软件这个整体承担的责任更少了。

我们把职责强行分成了某种层级，有些人考虑的是“整体技术方案”，还有些人只是思考什么 `if` 语句、`for` 循环还有标识符之类的，这就会把本来紧密联系的两个行当割裂开来。

纯粹的技术架构师可能构思的是最优的构架、最优的设计模式或是最佳实践什么的。但是，只有亲自上手，埋头在代码里的时候，他

们才能了解真正的难点在哪里。与此同时，那些无权做高层次构思的程序员也没有机会提出不同意见。常常是真正做实现的人才能最清楚地看到前面的障碍。

对于架构师–程序员这个例子已经谈得太多了。软件行业的公司阶梯需要一个更好的模式。

要盖一座建筑，设计师做设计，开发商做开发。传统的建筑师知道如何做详尽的规划，每个细节都规定得清清楚楚。但是他们不会自己动手去盖，因为这完全没道理。做高层次构思和在地沟中干活的这种分野，主要是根据两种角色的实际情况决定的。

软件行业就不一定是这样了。伟大的程序员可以同时在“地沟”和“高层”之间游刃有余。当然，架构师可能大部分时间都在做高层次的构思，但也应该对开发有所涉足，来了解整个的情况。

找时间写代码

在很多技术公司里，我前面提的这一条根本行不通。大多数技术架构师全天都在和公司里面的其他部门开会，他们还经常被拉去和客户打电话，讨论软件项目面临的种种技术难题。那到底什么时间去写代码呢？

在我的某个全职网络开发作坊创立几个月后，我们聘用了一个新的高级架构师 Adam。他来了以后，给我们年轻的网络程序员团队定下了一个非常不同的基调。除了所有日常的本职工作之外，很显然他对代码怀有激情。我马上就感觉到我似乎只是在和另外一个程序员对话——虽然他比我聪敏也睿智得多。我们之间的架构师–程序员关系让我个人也深深受益。

我们的第一个项目是给一家大的律师事务所做外部网。Adam 提到一些关于代码生成的事情，我听了觉得跟科幻似的。但是我很快就

发现，底层服务器端的对象、查询以及我要手写的那些方法大部分都是算法性的，很多都可以从外部网的数据库模式（schema）中演绎出来。Adam 建议我不要暴力开发埋头苦干，而是要重点建立一套定制表单和屏幕，他自己则着手写一个代码生成器。他每天利用坐火车上下班的一个小时写代码，连续做了几个礼拜。很快就有了一套虽然粗陋却很强劲的工具，可以生成本来需要手写的很多东西。

我们在代码生成上的小小分歧所浪费的时间，在开始使用代码生成器之后很快就弥补回来了。每次改动数据库模式，他就运行那个奇妙的小程序，重新生成继续制作软件所需的所有代码。很快，Adam 的努力所带来的价值超过了他写工具的成本。并且，这个工具我们还可以一直一直用下去。

所以，虽然我是项目的首席程序员，但 Adam 对开发也参与了很多。如果我需要一些不同的算法代码，他就在下班回家的火车上把新功能加到生成器里去。第二天，我就有了一套新代码，再也不用手写了。我在那几个月里学到的东西至今让我获益匪浅。

在你沿着程序员的职务序列，从程序员一步步变成架构师时，别忘了代码才是把这些角色融合在一起的粘合剂。可能不是坐火车的时间，却可以是工作中的一个小时、两个小时，专心致志，只写代码。你可以在第 23 篇中找到在编码时间心无旁骛的方法。最后我强调一点，无论你在开发团队中的职务级别如何，都要坚持写代码，这正是你最有价值的地方。

第 5 篇　扔掉旧代码

最近，我有个同事 Mustafa 说我“又在做这个”。我是在收纳代码（code hoarding）：把我觉得再也用不上的代码注释掉。虽然我们对所有源代码都有版本控制（你绝对也应该这么做），但我实在没有勇气把它们都删掉。因为我随时都可以找回那些旧代码，所以能直接删掉的时候，就没必要注释掉了。

收纳代码算是那种表面上看起来很正确的习惯之一。这是从其他基础工程行业里沿用过来的，却不太适合编程。如果要造一辆车，我们可能会把所有废铜烂铁都收起来，因为日后还可能用到，直接扔掉实在太愚蠢了。传统工程里面，人工和材料非常重要。现实生活中，在差不多成型的东西上修修补补比全部推翻重来要容易多了。

可是编程的时候，我们却容易太过强调这种因素。我们的工作真的是劳动密集型的吗？也不是。打字算不上是什么重体力活。那需要什么材料吗？我盘点的时候还从来没有发现“敲键盘”这个动作会有库存短缺。

与此同时，收纳代码实际上是制造了更多的障碍。多数时候的实际情况是，我从来不会把几天前或者几个星期前注释掉的代码再去掉注释。可我手头正在写的代码周围却满是这一大块一大块的灰色垃圾，看起来就烦。每次我干活的时候，周围的旧代码总会让我分神，没法专注在眼前最重要的事情上。

即使我确定要重新实现我前一阵子写过的东西，早先注释掉的东西一般来说也无论如何用不上。我可能把其他地方的逻辑动过了，旧代码里面引用的对象或者方法可能也变了。比起重新干干净净地把代码写对，要把这旧代码救活，我得花上更多的时间在那儿东敲西补。

上周的代码是“上周旧版的我”写的，那个我只知道上周版本的软件是什么样的。

不要把代码囤积在注释里，删除代码可以让代码库精简。眼前的页面应该精确地反映出软件现在的工作方式，一分不多，一分不少。现在就扔掉旧代码，在编程中间就不用跳过一堆不相干的垃圾字节。我们以后也用不着去琢磨，这一大团已经注释掉可看起来还很重要的代码，到底还是不是那么重要。

第 6 篇　多元化胜于专业化

在软件行业，我们可以身兼设计师、程序员和数据库管理员。一个人可以精通 PHP、Java、.NET、C++、Python、SQL，还可以了解 HTML、CSS、JavaScript 和 Flash。但似乎很少有人能够轻松地跨越用户界面和后端之间的界限。

在传统建筑行业里，让电工兼任水泥浇铸工，或者让铺砖工去装管道都是不现实的事情。他们都各有专长，各司其职。他们现实中也不会呆在一起。出于智力和实际考虑，这种情况就要求有一群专精的从业者去磨炼各自的手艺。

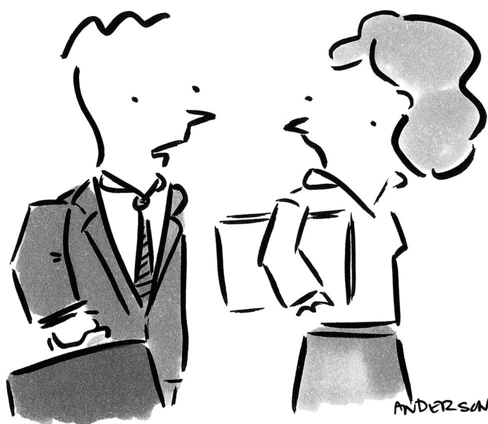
但把同样的理念移植到我们这个行业就不大站得住脚了。我们工作所用的工具无非就在眼前的这块屏幕上。如果正在搞 SQL，也用不着要换个地方才能写 HTML 或者是在 Photoshop 里面弄上一张图，只要在计算机上切换程序就行了。编程的科目之间，没有任何物理上的障碍。

此外，很多软件概念是超越语言，甚至经常是超越领域的。模型–视图–控制器（Model-View-Controller, MVC）是一种应用程序架构，被许多用户界面平台（如 Adobe Flex 的 Cairngorm 平台）采用，还用在.NET 等许多服务器端开发框架上。现在的编程语言之间有大量的重叠。设计模式和重构这些概念在编程的世界里随处可见。

在我的公司里，开发团队的大部分人员都了解多种编程语言，同时兼做前端和后端。这可以帮我们平衡每个人的工作量，因为大家都可以很熟练地在软件的所有层面上工作。

一个人可以同时是.NET 程序员、HTML 标准能手和数据建模专家。我们可能对这个或者那个精通、有兴趣，但没理由不能在多个领

域大显身手。



“我过去是做先进商业平台解决方案的，现在我无非是把东西搞定。”

为什么伟大的程序员不能同时是优秀的用户界面设计师呢？我经常听到程序员立马否定他成为杰出视觉设计师的哪怕一点点可能性。从概念上说，设计用户界面和设计坚实的软件架构差得并不远。出色而好用的用户界面要直观易学、组织清楚、扩展自如、目的明确。这和软件设计所推崇的许多特质是吻合的。

反过来也是一样。才华横溢的用户界面设计师中，认为自己有能力成为编程高手的太少太少了。可能程序员觉得用户界面设计是要让东西“漂亮”，而设计师看编程就是要写很多“技术玩意”。可是，他们的共性远比这个要多得多。

归根结底，无论是上层界面，还是底层引擎，软件设计的目标是一致的。我们没有理由不能在多个领域都做得出色。

第 7 篇　比喻渐欲迷人眼

我们已经知道了为什么比喻可能会影响我们处理软件的方式。如果过分强调比喻，就可能在这些不实的印象中养成一些坏习惯。比喻的害处还不止如此，它们不仅会降低我们做事情的效率，而且还会妨碍我们思考更好的方法来做一件事。

线框图和详尽的规划占用了实际构建和审查的时间，也不能利用我们能够不断迭代的优势。它们让人在一行代码都没有写的时候，就要把整个写代码的过程想一个遍。

对于发行的过分强调掩盖了一个事实，就是今天的软件已经可以比较容易地修改和重新发布了。我们不再“发货”，而只是从互联网上下载，或者软件本身就是基于 Web 的。要是发行日期因为某些必须要塞进去的功能推迟了，就会伤害到程序员的士气。这对于开发团队来说完全就是压哨绝杀。

软件开发中的传统职责分工，即架构师、程序员和项目经理的划分，压抑了那些在多个领域都有才华的人。一个人可以同时是有远见的规划师、周到细致的程序员和优秀的沟通者。严格遵循传统的比喻妨碍了真正有才华的人去接触这个行业中的各种机会。

要绕开这个问题，一种方式是找到一个更合适的比喻。开发软件可能和写小说或者作曲更相近。想想这些专业的人士是如何“规划”他们的工作的。

作家可能会列一个章回提纲，想想大致要写些什么。但接下来就马上着手写具体的东西，然后再反复编辑修改——这里动一个词，那里删掉一章。写作的过程和我们编程非常相似。

音乐家也不会一连好几个月都在写谱子，然后希望这谱子一次写

就。他会不断地弹奏、试验，然后找到一个合适的即兴片段。也可能会先有几行歌词，然后再配上正确的和弦，或者是反过来。歌是一点点写好的，然后一段段试验出来的。

在这两个例子里面，材料的成本都是很低的。纸笔都是现成的，吉他也不是按音符收费的，声音也没有成本。同样道理，代码是我们拥有的廉价材料。一旦我们把开发环境搭好，写代码没有什么材料成本。

所以，如果你面对一个软件问题不知如何下手，或者没有足够的信息来做出明智的决定，那么一开始用传统的开发比喻作为基础还是不错的。但如果你已经投入那个比喻一段时间，就需要抬头看看了，看看它在哪些方面促进又在哪些方面影响着你的流程。

知道了这些，现在就很清楚我们依赖的流程是不是真的有效。下一步就要着眼于长远了。如何在程序开发的职业生涯中一直干劲十足呢？接下来我们就来讨论几种保持干劲的方法。

第 3 章

动　　力

不管你有多牛，如果没有写代码的动力，那就拉倒吧。会计哪怕没什么动力也能把数据表填完，出纳也可以当一天和尚撞一天钟，但没有动力的程序员可真的能让一个软件项目彻底失败。

动力还必须是可持续的，在开发过程中必须不断挖掘，不断培养。项目初期让你写代码时干劲十足的那些因素，可能并不是项目收尾时的动力源泉。在制作软件的过程中，在不同的时间点，可能有不同的东西促使你前进。

保持动力的问题并不仅限于软件行业，在媒体上我们一直都能见到各种例子。签了好几百万美元巨额合约的大牌运动员，现在打比赛的时候不拼尽全力了；陪伴我们成长的乐队，现在开始粗制滥造发专辑捞钱了。即使有惊人的财富做保障，很多名人到后来也都疲沓了。这些都证明，仅仅一件事不足以维持动力。

我们需要多种方式来让激情在血管中不断奔涌。美食评论家的评论可以让大厨不敢懈怠，但餐馆生意红火、员工心情愉快也能做到这一点。好用的工具也有助于维持动力，如果给超级大厨一套高质量的刀具和一些新鲜原料，那就是见证奇迹的时刻了。开餐馆的动力来自于各种不同的方面，开发软件也是一样。

第 8 篇　工作即福利

在我们这个行业里，长久的动力并不来自于福利。当然，高薪和免费午餐确实不错，能随时玩玩桌上足球机也不赖。但归根结底，长久的动力来自于我们所做的工作。我所见到的每个有激情的程序员，在谈论起经过长时间苦苦思索才为技术问题找到的优雅解决方案时，都异常兴奋，这比说起在公司编程大会上赢得 10%的加薪要兴奋得多。

福利并不是长期的动力

正是出于这个原因，当一次又一次看到很多人，尤其是那些无忧无虑、只需月付房租的年轻人，仅仅为了稍高一点儿的工资和奖金，就勉强接受索然寡味的新工作时，我就感到十分困惑。



“我不知道，最近就是没啥能让我提起兴趣了……”

不要只因为加薪就耗在你痛恨的工作上，把这种工作思路留给那些就为了挣钱而编程，然后消磨时光企盼周末的人吧。

如果工资 X 和工资 X*1.05 的区别无非是一年中多在城里疯狂几夜的话，还是去做那些解决更有趣问题的工作吧，或者选择那些员工更有激情的工作。去那些有机会漂亮地做出东西的地方，在这里真实的项目才是大家的兴趣核心。你在工资上所牺牲的（如果你真的需要牺牲的话），将在幸福感上得到更多弥补。

好公司的一个标志就是它对待项目的方式。好的项目有明确而具体的目标。好的项目要么万事俱备，要么有做到万事俱备的计划。好的项目既有宏伟的目标，又经过周密的思考。好的项目有确定的交付时间，而不是时间和预算都含糊不清。这种项目给工作设立了一个目标。我参与的编程项目数以百计，所有好的项目都有这些给人动力的特质。

福利可能是毁灭性的

甚至有证据表明，表面化的福利实际上会削弱人们工作的积极性。是的，在我们面前晃动的胡萝卜可能让我们更加没有工作的热情。

《纽约时报》畅销书作者 Dan Pink 有一个很棒的 TED 演讲，是关于令人惊奇的动机科学的。[1]他说传统的商业激励因素，比如大笔奖金，可能会成功调动员工的积极性，但只能用于那些简单琐碎的工作，类似于把数据从一张表填到另一张表这样的工作。[2]

相反，那些需要批判性分析和创造性解决方案的工作，就像我们每天面对的这些，把金钱奖励挂在员工眼前晃悠则没什么用处。在一些涉及高层次思考的实验中，金钱激励和业绩之间是负相关的：给一组特定研究对象的金钱奖励越多，他们最后做得越糟。对于一个本已很有吸引力的项目来说，额外的金钱奖励反而让工作不那么诱人了！

① 参见 http://www.ted.com/talks/dan_pink_on_motivation.html。

② 如果想看原文，请参阅他的 *Drive: The Surprising Truth About What Motivates Us* [Pin09]。

对我来说，做的是什么并不那么重要，可以是只有一个页面的网站、搜索引擎、在线索引卡、交互性地图或是游戏。产品可以供几百万人用，几千人用，或是仅供八个人使用。我做的可以是持续六个月的项目，也可以是两个小时的练习。我可以大部分时间在写标记、做用户界面、写服务器端代码，或是搭建数据库。总而言之，不论在做什么样的工作，只要知道有机会制作优美的东西，我就会对这个工作很有激情。

在选择下一份工作的时候，记住那个能够让我们长期保持干劲的东西——不是外部的福利，而是工作本身。

第 9 篇　从喜欢处入手

有时候，最难找到动力的地方就是最开始。光是想想编码是很容易的，软件在我们脑子里面编译的时候都完美极了，我们不会纠结于将要面对成百上千个小障碍。可一旦真的动手写起来，就完全是另外一码事了，激情很快就会消退。

写软件的体会和写这本书没什么两样，我在琢磨要写什么内容时所花的时间比实际动笔的时间多得多。写作有时候真是一种吸人脑髓的游戏，其中充斥着了无生趣的字句，时不时还会思路卡壳或疲倦不堪。面对那些消磨士气的因素时，还想文思泉涌是很难的。

Natalie Goldberg 的 *Writing Down the Bones* [Gol05]一书从头至尾都在讲写作者的动力。她给了一个入手小提示，即不要去想什么宏大的开篇，而是从故事的中间找个地方开始写，从最有意思的地方开始，并且立即开始，不要试图从头写起。

我们总是花很多时间琢磨那个宏大的、抓人眼球的开篇，而实际上，这对于整个故事来说是相当无关紧要的一部分，写完开篇的那一段落之后还有无数的工作要做。我写这本书时就没有从头写起，我写东西永远都不是线性的过程。开始的时候，要是某个话题让我有了灵感，我就写这个话题。

这种思路也可以用在开发软件上。我们不一定非要先做首页，再做子页面，也不一定非要在业务逻辑之前先搭数据库。写软件不用非得一上手就从头开始，我们可以从最感兴趣的地方入手。这种优越性是造很多其他东西时所无法享受的，比如造房子、造汽车或者造任何实际的东西。我们并不一定要从某个特定的地方入手，许多东西都可以事后重构。也许这是绕了一点弯路，但如果一直灵感喷涌而不是苦

苦煎熬，我们就可以做得更快更好。

所以，如果你可以选择从什么地方开始写软件，一定要利用这份自由，挑一个你觉得最有意思的功能，从那里开始做。

这在着手做一个大型软件项目时尤其有用。用不着花上三天时间来框定一个进度表和发布日期，不如用这工夫写你最感兴趣的那部分程序。过一个礼拜，你就会知道自己到底有多少动力，对于其他部分何时能够搞定也会心中有数。

如果你发现自己就是三分钟热度，那么及时停止，损失也不大。不过大部分时候，你会发现写软件这种让人望而生畏的工作，其实也不是不能完成的。用三天（或者一个礼拜）踏踏实实地去写程序，你会对自己在做的东西以及剩下的部分何时能够完成有更多的了解。有了一点灵感并做出了一点点成绩之后，再去制定一个切合实际的时间表就容易多了。

第 10 篇　莫求全

每个有激情的程序员最关心的就是自己的代码。代码就是我们的画板。就算没有一个用户会看这些代码一眼，有激情的程序员也会一行一行地埋头苦写下去。即使我们知道，清清楚楚地知道，自己所做的不过是面向极少数人的一个小程序，但我们很多人仍然会关心程序在最广阔的舞台下表现如何。我们关心代码在最严苛的条件下的性能，并试图减少对服务器、对服务、对数据库的冗余调用。

然而，要在这个行业中生存下来，你最好不是个完美主义者，因为没有什么软件是完美的，特别是基于网络的软件。我们的产品是通过用户存活下来的。用户基数增长的时候，它会发生嬗变。新功能会不断呈现，新 bug 也会不断产生，所以要求完美实在让人筋疲力尽。

一个程序员最开始写下第一行代码时所采用的方法，和他今天所用的方法很可能大相径庭。软件会随着时间变化。我们编写、调整、反复，有时候还得重写，所以最好能适应这一点。

开发的时候，往往要做很多权衡，是追求性能还是追求代码简洁性，是要完美的构架还是要可维护性，等等。没有什么“高招”能够确定某种方式就一定是对的。

一个伟大的程序员会痴迷到有强迫症，但也一直都能接受不完美。想要写出“完美的代码”会让你骑虎难下。我们越早接受不完美，就越能保持干劲、坚持到底，最后完成的工作也会越多。

第11篇　休止一下

你可能是程序写得太多了。

当你全神贯注地工作的时候，当你的大脑完全沉浸在代码里的时候，当你的手眼脑在协同工作的时候，停下来。抬起头来，想想这一天的工作什么时候结束，盼着关上电脑，出去晃晃。



编程，虽然是一种脑力劳动，却也是一项舒服的身体活动。编程的时候一般都坐着，消磨了几个小时之后，我们在椅子里会越缩越低。有些人甚至一边编程一边吃喝，只要看看键盘就知道了——键面光滑玉润，下面却藏着几斤饼干渣。

这种舒适是很危险的。这意味着一做起来就是好几个小时，都没有意识到我们消耗了多少体力。

当代码开始变得有点拖沓的时候——或者，最好是在这之前——

就停下来。良好的编程习惯是在精力充沛时高效地工作，而不是一味地呆在屏幕前。两个小时的高质量编程胜过八个小时的煎熬。我们编程编累了的时候，更容易走个捷径，或是违反标准规范。那些低效的时间最后都写了些坏代码——我们第二天就可能后悔的代码。所以，请缩短编程时间，出去走走，享受一下生活吧。

第12篇　早起先测试

早起上班第一件事：测试你的软件。这是你最清醒、最有动力写点好东西的时候。

在一天的工作中，我们花了那么多力气去写代码，根本没精神再去挨段测试了。一天下来，会越来越难以统览全局。到了傍晚，人已经沉浸在这个软件中了。疲惫令我们无法判断什么有道理或是直觉上正确，还会让我们漏掉一些细节。

这个功能应该放在这儿还是放在那儿？这个函数要不要换一屏？这么做有道理吗？刚做的调整真的那么重要吗？到下午五点时（对于那些真正工作过度的人来说，是到了凌晨两点时），我们已经很难判断软件给人的感觉了，因为鼓捣它太久了。

然而，在早上九点，一觉醒来，精神焕发，我们常常可以更好地回答这些问题。前夜一团乱麻的思路已经烟消云散。在一头扎进代码之前，这是再度审视软件的最佳时机。

伴着朝阳，我们对软件又有了新的感觉，对它的看法也不会受到后台实现的羁绊。因为已经离开它有一段时间，所以我们可以用一种更公正的眼光审视它。

早晨，我们会忘记前夜困扰我们的繁复的代码细节，也不会纠缠于某个功能不太精彩的实现，我们会全心投入眼前所见的东西，而不去思考它背后发生的事情。



“这些是我在你桌子里找到的……”

测试的时候，从头开始。不要纠缠于某个部分，重新体验一遍就对了。前夜，你可能在实现一个功能，但真正的用户可能只会使用它一两次，甚至从来不会去用。早晨，要专注于那些大多数人会经常使用的功能。关注软件中的优先事项，关注那些需要首先修正的东西，这种方式要好得多。

在编写新代码之前，要在早晨，即你最清醒的时候先做测试，这可以保证你一直在开发好的软件。

第 13 篇　别在卧室里工作

20 世纪 90 年代末，我就开始在大学宿舍里面建网站了。到了大四，我的时间基本上就分成两块：白天上几节课来凑够学分，晚上就在宿舍里面搞自由职业——设计网站。作业？什么作业？

当有些同龄人在食堂或者计算机实验室（还记得吗？）做兼职挣最低工资的时候，我却穿着短裤坐在屏幕前，动动鼠标就能拿到五倍于他们的小时工资。我离床铺只有一步之遥——要是某天过得不好，走过去也许要两步。

有段时间，我觉得生活就该这样了。我一边拿着学位，一边还拿着对于一个在卧室里工作的大学生来说非常可观的薪水。

毕业之后的那个秋天，我去了芝加哥的一家小型技术公司上班。当时正好赶上互联网泡沫破灭，美国经济一落千丈。工作了六个星期之后，我的职位也“破灭”了，于是我回去和父母一起住了。一夜之间，作为一个 22 岁的成年人，我又在舒适的卧室里做起了自由职业者，接了几个网页设计项目。

头几个星期的感觉棒极了，但起初的蜜月期过后，穿着睡衣工作就觉得有点别扭了。毕竟我不是在学校了，朋友们也不在我楼下或是校园里了，周围就只有我和我的电脑。由于没有其他必须参加的活动，比如去上课或是拿学位，所以我所有的时间都用来工作了。

我不是每天连续挥汗如雨八个小时，而是一天到晚都在断断续续地工作。早上先专心干几个小时活，然后吃点点心，看上几个钟头的日间法庭电视节目，再干上几个小时，之后跑个步，吃个饭——哦，对了——晚上还要再干几个小时。就这样，原本四五个小时的收费工作时间被摊到十二至十四个小时里。我的工作和生活混为一谈，工作

的激情和效率也没了。

正如帕金森定律所说："工作会不断膨胀，直至占满所有可用的时间之后才会完成。"由于我可以在一天中的任何时间"工作"，所以可占用的时间就多了去了。每周 40 个小时的工作突然就变成了 168 小时的工作+睡觉+吃喝娱乐。

在家办公是一种奢侈。大部分人都宁愿在家办公，也不愿花两个小时去上班。但如果你有幸享受这一点，千万不要在卧室里办公，最好也不要在起居室。搞出一个封闭的工作区，最好是单独的一个房间，这样就可以在工作时间结束后从那里离开。一天结束之后关上门，挂上一个"打烊"的牌子，然后去享受生活中的其他乐趣，第二天再继续工作。

这才是真正的生活。

第14篇　第一印象也就那么回事

对软件来说，用户的第一印象到底有多重要？尤其是不那么好的印象？我一点儿也不觉得第一印象特别重要，我们不应该因为不好的印象而让自己的热情瞬间消退。

诚然，不好的第一印象可能是个信号，它说明我们的软件确实有什么不对劲。但我发现，造成很多不好的第一印象的原因有两个。

第一印象不好可能是因为不熟悉

有时候，第一印象不好完全是因为之前没用过这个软件。对待这种印象时，我们要持保留态度。比如，第一次用 Gmail 的时候，我心想：

> 这些邮件就像小型论坛一样，像讨论的线索一样……而不像电子邮件，有意思，也有点怪。我喜欢吗？不怎么喜欢。还行。我也不知道……也许吧。

Gmail 的邮件线索在最开始是个很奇怪的概念。我听过很多人对它赞不绝口，但也有同样多的人把它批得体无完肤。可几个月之后，我完全听不到这种争论了。今天，你和用 Gmail 的人之间的谈话更可能是这样的：

戴红帽子的人：嗨，哥们，你用 Gmail 吗？

戴蓝帽子的人：用啊。

戴红帽子的人：你觉得怎么样？

戴蓝帽子的人：还成吧，最近有点慢。不管了，我们去喝杯啤酒吧！

最后我发现，两种电子邮件系统对我来说都不错，对上面那两位

来说也不错，对世界上绝大多数人来说都不错。过了一阵子，我们就都不再纠结了。实际上，现在我同时用 Outlook 和 Gmail。我也知道很多人同时使用非 Gmail 的邮件客户端和 Gmail。我用 Outlook 的时候，看到的就是普通的老式邮件；用 Gmail 的时候，看到的是“类似于论坛的邮件形式”。到头来，两种感觉都挺舒服的。



“我是接受比较慢的那种。”

我承认，重塑像电子邮件这种深入人心的应用模式确实需要勇气。不过幸运的是，在我们这行，这种风险要小得多。单选按钮还是下拉列表？每一页都有搜索框还是只有一页有？最可能的回答是？都行，都行，都行。到头来，只要我们习惯于一遍又一遍地看同一个软件，很大的可能是，不管最开始有多么不喜欢这个设计，最后都会适应。

我知道那些反对者正在大声抗议。我的意思真的是用户应该适应软件，而不是反过来吗？我真的是在说，对于软件的第一反应不重要

吗？说这话的是那个写了 *Flash Application Design Solutions: The Flash Usability Handbook* [CB06]的家伙吗？是的！

我不是说第一印象毫无价值，但第一次见到某个东西的时候，最初的反应通常不过是对不熟悉事物的反应而已。人类天生会对新事物心怀畏惧。但是，作为开发人员，我们常常太把用户的第一反应当回事了。

第一印象不好可能是因为关注了次要问题

另一个问题是这样的：有时候，不好的第一印象并不能反映出真正重要的问题。

如果 Google 刚开张，而我是一个可用性测试员，那么我的第一印象可能是这样的：

- "Google 搜索"按钮应该和"手气不错"按钮调个位置，因为我提交信息的时候习惯于点最右边的按钮，而且一般我会真的去搜索，而不是碰运气。
- 我不明白"手气不错"到底是干嘛的，看了就晕，应该有个提示告诉我点了以后会发生什么。
- 我找"高级搜索"选项可找了一阵子。哦，那个高级搜索页面真难用。
- 底部的导航链接应该是在搜索栏上面，因为我习惯于导航栏出现在那里。

要是现在再问我，最初的那些疑虑大部分都可以消除了。我已经适应了搜索按钮的位置。用了几次"手气不错"按钮之后，我现在知道它无非是直接把你转到第一个搜索结果。高级搜索功能？我从来都不用，所以管它好用不好用呢。那些底部的链接也是，我很高兴它们是在搜索框下面而不是上面。

第一印象常常是不准确的，因为我们不知道最后是否会适应这个软件。最开始觉得重要的东西，比如高级搜索功能，可能压根儿就不重要。对于那些和我们的习惯稍有出入的东西，比如“搜索”和“手气不错”的顺序，过一阵子就适应了。

所以，当你从顾客、客户或是同事那里得到负面反馈的时候，要坚持自己的想法，解释你这样做的道理，让他们适应你的作品几天，而不要让最初的即时反馈打消你的积极性。

如果问题仍然存在，那么你的软件可能确实有瑕疵，但你会惊讶地发现，很多最初的负面印象通常都消失了。

第 15 篇　软件发行的情感价值

上一章中我谈到了发行。在如今的开发过程中，发行无非是软件生命周期中众多次发布中的一个。但发行之所以重要还有另一个原因：发行日期本身是一个强大的动力源。

为啥呢？因为我们知道软件不再是在局外等待了，它有生命了，准备好了，这对自信心是一个巨大的提升。

第一期完工带来的美好感觉对于我们后续工作的效果和效率有巨大的影响。想想无穷无尽地调整那些还没能闪亮登场的软件时的丧气感觉吧。

不过有时候，我们害怕发行，因为发行就意味着我们珍贵的软件要任凭公众“摆布”了。他们会怎么说呢？就像上一篇说的，他们最开始说的可能并不是日后他们所想的。即使必须做出些关键的改变，我们也做得到。很少有功能加减或逻辑调整是不可能实现的。好的程序员一直都准备着干这个呢。设计模式、重构方法和最佳实践主要都是为了适应未来修改的。

所以，能发行时就发行吧。下一章谈到生产率的时候，你会看到保证发行日期不会一拖再拖的方法。

第 16 篇　找个争论话题

除了去写下一个伟大软件的代码，还有其他的方法来保持动力。

找个争论话题吧。找一个你无比赞同的话题，要是找一个你毫不掩饰地反对的话题就更好了，然后大声讲出来，细致入微地解释为什么你的方法行得通。

参加一个当地的 Meetup.com 讨论组，或是申请在会议上发言。不要觉得你水平不够或是准备不足，如果能找到一个热爱的话题，你就已经准备好了。你不需要是一个拥有五万名微博粉丝的技术明星，也不必是来自一个超级成功的公司的编程英雄。就现在而言，你只要做自己就好了。网络社区的好处就在于，你的发言权首先是建立在内容上的。如果你有话要说，人们会想听的。

如果你一想到发言就手心冒汗的话，那就写写吧。写作让你有机会字斟句酌，让它臻于完美。你可以自己建个博客，也可以联系其他有影响力的博客作者，写上一些客座文章。你会惊讶于这个社区有多么包容。

需要争论话题？下面就是几个观点两极化的话题，你可以从这里入手。

- 在 HTML 标记中，对于表格内容之外的东西用`<TABLE />`合适吗？很多标记语言纯净论者说“不”。也许你可以说“是”，然后解释为什么。
- 在富网络应用中，Adobe Flash 还有意义吗？或者，HTML5、CSS3 和 JavaScript 的进步是不是足以淘汰 Flash？
- 对象关系映射（ORM）比原生 SQL 更好用吗？很多人认为 ORM 在做复杂数据库查询时通常效率低下，其他人则认为

ORM 为开发人员提供的简洁性足以弥补该不足。

- 模型–视图–控制器（MVC）架构是不是构建所有应用软件的最佳方法呢？有些人说这个架构太过臃肿，单单使用标准页面模型更好。
- 可用性测试和 A/B 测试对于网络应用到底有多重要？有人说它们言过其实，对应用进行研究所付出的先期成本和时间是不值得的。

正如本章所说，动力可以有很多种形式。它不一定只局限在代码里，它还可以是我们离开办公桌时看待工作的方式，甚至一个好的争论话题也可以让我们热情高涨。

如果你能够在整个开发生涯中保持动力，那么你的效率也会提高。让我们来看看如何把动力转化为持久的生产力吧。

第 4 章

生　产　力

动力可能是起步时所需要的，但生产力才是衡量成功的具体标准。本章讲的是如何保持干劲，可以每天都激情饱满地工作。

在企业里，生产力和其他任何概念一样，都换算成某种可以计算的指标。就生产力而言，这个指标就是利用率（utilization）或者产能（throughput）。生产力常常用一个公式来表示，计算我们做了多少工作，或者我们同时在应付几件事情，而不是工作的质量如何。真正的生产力其实应该关心工作的质量。

比如同时处理多项任务，让我们感觉自己似乎很高效，但这么做很少能真正把工作做好。“边吃午餐边工作”就是一例，一只手拿着老长的三明治，另一只手用一个手指敲键盘，这样能写出多少高质量的代码？

更重要的是，这么干实在不怎么舒服。请走出办公室，从从容容地吃顿午饭，然后出去呼吸呼吸新鲜空气。等你回来的时候，代码就得来全不费工夫了。

第 17 篇 对消闲项目坚决说不

我们每个人都有一堆消闲项目，可能是写了一半但从来没完成的软件，也可能是开始写时干劲十足，但因出现更紧迫的事情而戛然而止的代码。可以怪其他工作占用了时间，也可能是我们自己已然失去兴趣了。

这类消闲项目本身没有严格的时间限制，而且不成功也没什么关系，因此注定要失败。如果它的发行日期不定，只是“未来某年某月的某一天”，很可能我们近期就不会去完成它。有些人似乎花了几年时间来琢磨下一个伟大的构思，却没有一开始就明确具体要花多少时间来实现它。

三个月怎么样？Jack Dorsey 开发了一个鲜为人知的短信服务，他从构思到发布第一版所花的时间还不到三个月，这个服务就是后来的推特。[①]试想一下，如果他年复一年地开发，而不是一旦开始就闪电发布，那么结果可能就大不一样了。

时机就是一切

由此可见，时间是保持编程动力的最重要的因素。对于消闲项目来说，刚开始写代码的时候，你可以出于好玩或者学习的目的。但是，当你决定认真对待它的时候，就需要定下一个期限。如果能正确处理下面几个问题，就可以把它变成真正的项目。

- ❑ 我每周要花几天、每天要花多少时间在这个项目上？
- ❑ 我什么时候能把一个基本成型的产品展示给别人？

---

① 参见 http://en.wikipedia.org/wiki/Twitter。

- 哪一天向公众发布？
- 哪一天发布第一个主要的更新版本？

第一个问题设定了可持续的每日工作量。可能只是一天两小时、一周三天，但必须要坚持。每天下午三点到五点做它，比说什么“有空就做”要好。还要均匀地分配时间。每周一、三、五做它可能比周末一口气连做三天要好，否则两次之间的间隔太长，你要花好多额外的时间来回顾上次的进度。最后，时间表要切实可行。一旦养成“今天放一放，明天再补上”的习惯，你很快就会对它失去兴趣。

第二个问题为可供测试的软件设定了一个期限。这就像在地上打了一个桩：不久后的某个时间，我们的同事、朋友或是爱人就能看一眼我们的成果。这可以帮助我们倒算出，从现在到那时，一共需要多少时间来完成它。再考虑每天和每周能花多少时间，我们就能知道每次要写多久代码。

第三个问题让我们准备好一个“够用”的软件，在大的方面都无碍，可以供公众使用。它随时都可以发行了。从上一个日期到这个日期之间的这段时间都用来修复关键 bug，而不是添加成百上千个小功能——虽然我们很想添加进去，但几个星期内没有这些小功能的话也无妨。

最后该考虑第四个问题了。它安排了发行之后的一切。软件已经发行了，现在需要设定一个时间来推动新的发布。新的发布可能在发行一星期之后或更早，对基于 Web 的软件，可以是发行之后的几天，甚至几小时。一旦到了这个阶段，我们就可以不断发布新版本了。

这些时间限制就像建起了围墙，我们要用工作填满它。它帮助我们确定最重要的功能，为我们投入软件的每分每秒提供了目的。如果没有严格的时间限制，我们可以磨蹭一辈子，琢磨着是不是万无一失了。我们不是为交付做好准备，而是每一步都精雕细刻，调整个没完。

失去了那种紧迫感，生产力就会枯竭。时间限制敦促我们前进。

设定一个最后期限，即使是随便设的

我们公司的第一个正式产品 DoneDone[①]，就是从我的一个消闲项目发展起来的。DoneDone 是一个简单的基于 Web 的 bug 跟踪工具，侧重清晰简洁的工作流程而不是复杂的功能。我最开始做这个消闲项目，是因为不喜欢我们正在使用的那个每个月要付 120 美元的 bug 跟踪工具。有些地方我们本想简洁，可它却搞得花里胡哨；我们的流程本可以效率更高，可它偏偏缺点什么。我知道我能做得更好。如果我们愿意花钱买现有的 bug 跟踪器，别人也肯定愿意花钱买我们的。

开始几个星期，我只是在构思，没有线框图也没有说明书。我只是写代码、做界面、测试、调整，然后再继续写。我还处在开发蜜月期，虽然没什么方向，但一想到产品能挣钱就干劲十足。

几个月后，到十一月时，客户的工作又开始多了，自然而然我的消闲项目被排到了后面。每隔几天，我就抽出几个小时来做 DoneDone，但效率很低，因为大部分时间都花在重新熟悉前面所做的东西上。由于时间是零散的，所以突然闲下来的时候，很难决定具体应该做什么。

所以，我需要一种新的方式。作为一项业务，DoneDone 需要有客户项目一样的紧迫程度。它和我们为其他客户所做的项目有何区别呢？无非是我们是自己的客户。和客户项目一样，我们也需要制定几个期限：一个内部发布 DoneDone 的日期，一个向公众发布产品的日期，以及后续发布更新版本的日期。

最后，我决定要在 2009 年 4 月 15 日发布 DoneDone。每年 4 月

① 参见 http://www.getdonedone.com。

15 日也是美国国税局征税的日子，这二者间的巧合是否有些诗意？老实讲，这完全是随便挑的日子，大概还有六个月，感觉是个合适的时间——剩下的时间不太多也不太少。还有很多工作要做，但如果我把工作日 50%的时间花在 DoneDone 上，然后时不时找几个人来帮忙，就可以让这个项目运转起来。一个消闲项目就这样突然变成了真正的项目。

最后期限定好，就可以着手来完成必不可少的工作了。我们需要加入一个支付网关，确定成本结构，构建营销网站，然后清理功能列表。所有的事务都分配了时间，找到了位置。这种紧迫感——和生产力——又回来了。

软件发行之后，肯定还有其他的功能要加入。现在回过头来看，很难想象如果没有这些功能 DoneDone 会怎样不堪。我们当时没有针对问题的电子邮件-工作单系统或是标签系统，这两个都是当今系统的核心组件，但在发行时它们并不那么关键。我们把精力放在一个花六个月构建的 bug 跟踪工具所需的最重要的功能上。接下来的 90 天内，我们连续发布了十个更新版本。

我竟然写了一整篇文章来谈最后期限吗？还真是。虽然它一点都算不上创新，但制定了最后期限，工作才得以完成，否则产品永远难见曙光。最后期限提升了工作的重要性。如果让一个项目从几个月拖到几年，你的产品可能就失去开始时所期待的价值了。

最后期限创造了一种紧迫感，敦促你冲过终点线。即使没有人在逼迫你，它也能给你所需的鞭策。

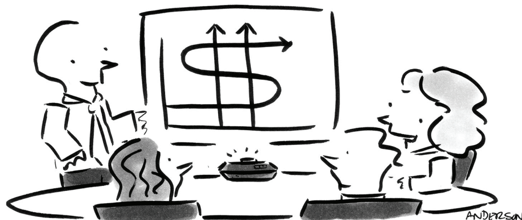
第 18 篇 限制所有的因素

在上一篇文章里，我们看到了时间限制可以把消闲项目变成真实的项目，但我们绝不是只能够限制时间这一个因素。所有软件都需要花钱来做，所以还可以为成本设一个上限。成本上限赋予我们创造力，它可以帮助我们更有效地利用资源。

想想那些彩票赢家的悲惨故事吧，他们本来朝九晚五地工作，可突然有了一大笔钱，自己都不知道该怎么花。他们没有把钱存起来，而是肆意挥霍，去买什么游艇之类的。他们把成千上万的钞票送给八竿子打不着的亲戚。过不了多久，许多人就会债台高筑，日子过得比中奖前还要惨。他们没有意识到，一百万美元仍然是有限的。虽然金钱上的限制变宽了，可限制还是存在的。

同样的原因导致了那么多风险投资支持的创业公司失败。在最初的互联网泡沫年代，一个风投把五千万美元扔给一群有着飘渺想法却不一定会成功的家伙，简直是稀松平常的事。有了这么多钱，人们很容易认为下一步就是在市中心租个宽敞的办公室，雇用几百个刚毕业的大学生，再成立一个董事会。毕竟，如果风投给你这么多钱，你最好还是整出点动静。

既然市场对未经检验的公司估值如此之高，想法的价值，而不是工作的价值就开始膨胀了。没必要先弄个小产品出来，放到市场里看看是不是有前景。调整几个功能、重新部署软件也不值得再加上一百万美元。有些公司不是根据客户的数量来确定 Web 服务器的容量，而是直接先买下整个服务器集群，然后坐等客户群增长。



“我喜欢！我太喜欢了！”

取之不尽的银行金库让人满脑子只有成功，而没有危机感。很多公司也貌似找到了富有生产力的招数，不过却没有围绕着自己的产品。

Kozmo.com 是一家风投支持的公司，其提供的服务是把食品和娱乐产品送上门，但不收送货费。要是只有几十万美元来运营，它也许会有压力早一点儿扭亏为盈。可是，Kozmo.com 烧掉了几亿美元，它把毫无利润的业务拓展到了美国九个主要城市。[①]到了第三年年底，Kozmo.com 破产清算了。由于起初就无须担心要开发一个能赢利的产品，Kozmo 从来就没有机会把一个有前景的想法转化成实实在在的产品。

如果没有限制，无论是时间限制、成本限制，还是功能集限制，我们就会忽略现实，做出有问题的决定，因为没有东西敦促我们做出明智的选择。生产力也就不会放在真正重要的东西上。

如果想要开发伟大的软件，请千万要设定并遵守限制。让你走的每一步都朝着创造更成功的软件的方向，因为没有多余的资源去做其他的事情。

---

① 参见 http://money.cnn.com/galleries/2010/technology/1003/gallery.dot_com_busts/9.html。

第19篇 去掉时间表中的细节

我搞软件开发十二年了，从来没见过一个项目是完全按计划走的。

功能会变，未预料到的障碍会出现。有时候，我们预计花一星期的事情最后花了三个星期。不过，最常见的事情是我们把项目时间表制定得过于详细。给每个小部件都设一个最后交付日期，会让我们成为时间表的奴隶。我们在一步还没走的时候，就决定了每一步要花多长时间。其实在最开始的时候，根本没办法制定出一个完美的计划。

所以，开始做计划的时候，要少计划些细节。

把时间表里的交付成果分成几个可以伸缩的大块，而不是小的面包丁。比方说你是在规划一个八个星期的项目，那就找出八个每周要交付的成果，而不是四十个每天要交付的东西。不要为软件中的每个交互动作设定交付期限，而是确定何时完成一个完整的部分。说到底，时间总量是一样的，但中间的检查点要少一些。

如果时间表太详细，交付太频繁，开发过程中就没有做试验或是重新考虑细节的余地。我们只能严格遵守根据设想出来的任务所设定的时间表，就好像被一个无知的“微观经理”一直监督着一样。要是有几个小任务没按时完成，整个时间表就轰然坍塌了。这一点也不会激发我们的积极性，好的软件不是这么开发出来的。

前端开发人员处理屏幕设计的方法可谓迥异。有些人喜欢在Photoshop里面做好一个完美的模拟，然后再把效果转化为HTML和CSS；有些人喜欢直接上手写代码；有些人喜欢先专注在标记的结构上，然后再加入个别的颜色或者调整字体大小；有些人需要一边搭框架一边设计；有些人先建好最底层的公用基础，然后再添枝加叶；还有些人则是先为最优的屏幕分辨率设计好，然后再优雅地降级。

为完整的屏幕设计设定一个交付日期是一个不错的里程碑，但为所有的中间部件（单为 HTML 或单为 CSS）都设定期限可不大好，因为这让开发人员失去了自由，无法选择自己最高效的工作方式。

在各交付物之间留出合理的时间，以便我们施展身手。可以把一个大型项目分割成多个微型项目，但我们仍然有机会以自己想要的方式来进行开发。这也让我们可以在下次交付前反复雕琢几次。在下次交付前留出一两个星期，也就允许我们犯几次错误，但仍然能够按时完成。

第20篇 每天改进产品的两个方面

咱们也不要把编程说得太美，因为我们时不时就觉得开发很枯燥。有些时候我就想干点别的，什么都行，给濒危动物作开胸手术都行。

在编程最热火朝天的日子，我们只能从那些很小的成功中获得持久的生产力。写代码本身带来的简单满足感或是成名的幻想已经没办法一直激励我们了。每天都得有那么一丁点儿灵感——平常的动力源泉枯竭的时候，这算是一个底线。

公司成立的第一年，我花了好多时间来做一个基于Web的数据建模应用，叫做X2O。直至今日，X2O仍然为我们在We Are Mammoth所做的每一个Web项目建立应用框架。它生成一个界面来建立应用的数据模型，并维护数据库、数据访问层和Web服务，这样我们就可以很快地开发基于数据库的定制应用。

X2O是一个有着宏伟目标的应用。它有些地方复杂得让人发疯，背后需要做无数的工作。拆开来看的话，它综合了几十个程序，生成定制应用的不同部件。开发这么个东西又要保持热情不是那么容易的，因为这几十个程序的每一个，本身就都是很大的软件。开发这种大家伙需要很多小小的胜利。

当我深陷在开发中时，我为X2O的日常工作设定了一条新规则：每天要对软件作出两点改进。

这不一定非得是重大的改进，也可以是很小的改进，比如在用户界面上添加更优雅、更友好的错误信息，或是去掉废代码，即便是保证所有的方法都有很好的注释也是一种改进。有时候，我一天能处理一大一小两个任务，有时候我可能就处理两个小的任务。不管怎么说，

每天我都知道，当天的 X2O 比前一天要好，而且这种进步是可以量化的。

“二”这个数字本身也很重要。做一个改进有时候让人为难，因为我们不知道到底选哪个；而且“一”跟“零”也没有多大差别。此外，“一”很容易让我们觉得，今天不做也没什么，可以明天再补上。反过来，每天做“三”点改进又太多了。所以，“二”才是那个神奇的数字。

决定每天改进产品的两个方面是一个很好的想法，它可以保证大型项目不断向前推进。一周下来，你的产品就会有十项改进；一个月下来，你的产品就有四十项改进了。

第 21 篇 为良好的工作环境投资

有没有注意到便宜的保鲜膜和贵的保鲜膜之间的区别？对于节俭的人来说，为一卷塑料膜付出双倍的价钱实在不值，但两种保鲜膜的区别太明显了：好的保鲜膜拉出来很容易，而且它不太会自己粘成一团，却会服服帖帖地粘在碗上。更重要的是，好的膜也容易撕下来。

在超市里，我们倾向于选择便宜的牌子，毕竟只是保鲜膜而已嘛。我们起初是省下了几块钱，但在每次撕下那片拉长了又不粘的塑料时，都要郁闷半天。

在厨房里做的每一件事都跟撕保鲜膜类似。要是每件工具都不那么称手，那么烹饪的体验就会充斥着一个个效率低下的环节。每次完成工作时遇到的小障碍越多，我们的效率就越低下。

同样的道理对我们程序员来说也特别适用。生产力依赖于我们工作环境中的每件小事。我们的工作环境应该尽可能减少这种令人分心的事。

速度快、功能多的机器会物有所值

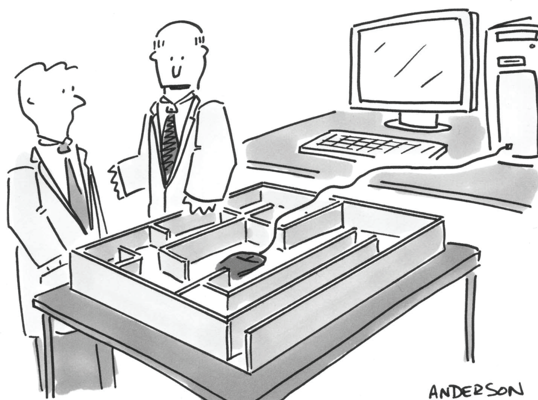
这就是购买好的硬件至关重要的原因。我们现在付出的金钱成本，会在每一天以生产力的形式偿还给我们。

我最近从一台用了七年的运行 Windows XP 的 Dell Inspiron 9300 笔记本，升级到了同时运行 Windows 7 的 MacBook Pro。这可是一笔不小的投资，但有了这笔一次性的投资，现在我工作的每一分钟都在收获它带来的好处。

我的旧笔记本也没什么毛病。我可以花一百美元升级一下内存，然后再勉强用上几年。但是，在许多细微之处，它就和那卷普通的保

鲜膜一样。我每次编程的时候都要付出小小的代价。

有了新的笔记本，我现在可以同时使用两种平台。必要的时候，我一边在 Mac 上运行快得多的 Photoshop 版本，一边在 Windows 上用 Visual Studio 做.NET 程序。另外，我还可以同时在 PC 和 Mac 上做浏览器测试，而不用在那些淘汰的电脑（很多公司通常标为“浏览器测试机”）上装好多个浏览器。



“还是没反应。你确定这是我们最好的电脑了？”

好处还不仅限于软件。我甚至爱上这键盘了。按键又平又薄，触感反应刚刚好。比起传统的笨重键盘，我打起字来更流畅顺滑了，连错字打得都少了。

想想每次不打错字意味着什么吧。这意味着思路被少打断了一次，也意味着不必将目光从屏幕转移到键盘上来重打一次，还意味着不必接续刚才中断的思路。

假设有了新键盘，我每天少打四次错字（我肯定现实中要比四次还要多得多），那么一年里面，我在写代码时就少被打断大约 1000

次，而为此支付的成本不过是一个新键盘。

加大“地产”投资

在厨房里，台面大总不是坏事。我们要是做一道需要几十种原料的菜，并且需要同时使用几个大器具，那么有一大片地方就太重要了。这样我们就不用把东西摞起来，而是可以根据自己的喜好把工具放置在不同的地方。

更大的台面，意味着不太可能不小心把生的原料放在熟的上面，把面粉袋掉到地上，或是因为到处找盐而把鸡蛋煎过头了。

在编程方面，屏幕上“地产”的价值也是这样。要是只有一个显示器，我们就得做出妥协。由于没有足够的地方来把开发环境、浏览器和通信工具客户端同时“摊在台面上”，我们每天不得不在不同状态之间切换一千次。

假设我们想在测试环境中严格测试代码，同时在浏览器中运行编译好的应用。要是屏幕不够大，我们一次只能看一个应用，或者两个应用都缩小到细细的横向滚动的小窗口里。这种小调整实在是让人分神，可能会扰乱我们的思绪。

就像厨房台面一样，多个显示器能大大提高生产力。

如果有两个显示器，把开发环境摆在面前，全屏显示，另一个显示器上则一直开着测试浏览器，所有其他程序（比如邮件和聊天客户端）也放在这里。这样，你就可以一直专注于开发，并且在另一块显示器上做测试。

要是有三个显示器，可以把开发环境放在面前，另一个显示器开着测试浏览器，然后把所有其他程序（比如邮件和聊天客户端）放在第三个显示器上。如果你想要全神贯注几分钟，不想让闪烁的任务条图标和未读消息分你的神，只要暂时把第三个显示器关掉就行了。

说个题外话，下次找工作时，若想知道该工作好不好，可以迅速扫一眼办公室，就知道管理层是不是关注开发团队——是不是真正关心开发人员的工作环境。数数房间里面的显示器，然后除以员工人数。得出的值就是“关注系数”。

关注系数	诊断结果
1	不关注
1~2（不包括2）	有一定程度的关注
2~3	非常关注
大于3	你进入短线操作者办公室了，马上走人吧

工作了几年的环境可能现在看起来还行，也许你已经习惯了，但多花上几千美元可不光是一笔开销，而是对生产力的投资。

第22篇 列一张个人待办事项清单

有时候，最简单的工具可以让生产力大大提高。列一张个人待办事项清单吧。

个人待办事项清单*不是*（我再重复一遍，*不是*）像项目时间表或甘特图那样。那些文档都是给团队用的，对个人来说太笼统了。它们都是反映项目的大局，而不是指引下面要进行的几步。当然，它们作为“大局”文档很有用，但却不能帮助我们理清头绪。

个人待办事项清单也*不是*爆满的邮件收件箱。用电子邮件来提醒自己有什么事情要办是徒劳的。电子邮件是一堆零碎的对话、问题和未区分优先级的任务的大杂烩，而不是一张现在要做的事的清晰列表。电子邮件也不是用来快速浏览的。

个人待办事项清单*只是一张检查单*，仅此而已，快速、简洁，而且有效。

当邮件里收到一个任务的时候，请把它记在个人待办事项清单里。如果加入了一个必须具备的功能，把这个功能拆分成小的事项，加入待办事项清单。

乍看起来，个人待办事项清单无非是又一个要维护的东西，里面所有的事项很可能都来自于其他的文档。这似乎是个不好的做法，因为它违背了大多数人在写代码时遵照的“不要重复”的箴言。然而，个人待办事项清单是少数可以接受重复的例外，这是因为，它的作用和其他更为严谨的文档不同。

和其他文档不同，待办事项清单是随时可以调整的，从来都不是一成不变的。待办事项可以每天添加、勾掉、推迟、提前或是删掉。待办事项清单与项目时间表和甘特图不同，它不关心过去，它的起始

点永远是现在。它列的也不是猜测的任务，而全是井井有条的、真实的、必须在近期完成的任务。

好的个人待办事项清单的内容

程序员的个人待办事项清单应当具有下列特质：

- 它是一张清单，且只有一张清单。
- 它有四个栏目，即今天、明天、后天和未来。
- 它没有嵌套的依赖关系。每个待办事项都直接属于上述一个栏目。
- 容易修改。可以很容易地把事项挪上挪下。
- 由若干个短任务构成，在几个小时之内就可以完成。未来一栏中的项目可以更宽泛一点，我们后面会谈到。
- 在线。不管在哪儿工作都能看到。

我自己使用 37signals 的 Ta-da List 软件，因为它简洁又免费。下面就是用 Ta-da List 创建个人待办事项清单的步骤。

创建一个新清单，然后加入四个分割项目。因为你不能在 Ta-da List 里面建立分隔符，所以只要放进四个永远不会勾掉的待办事项就行了。事项前后加上短划线，这样容易和其他真正的待办事项区分开来。

我的个人待办事项清单

- ☐ - - - 今天 - - -
- ☐ - - - 明天 - - -
- ☐ - - - 后天 - - -
- ☐ - - - 未来 - - -

添加项目

在添加待办事项时，把它们放在合适的分隔符下。如果一件事你必须今天完成，就把它拖到今天下面。如果是需要明天处理的事情，

就把它拖到明天下面。如果需要后天完成，就放到后天下面。如果你也不知道具体要什么时候完成，但是知道很快会有一个任务要来，就把它放在未来下面。

因为待办事项清单中没有什么是定死的，所以如果你不确定应该归在明天还是后天，那就选明天。如果到明天结束，它还没有成为最紧急的事情，那就可以再推一天。

把功能拆分成待办事项

你放进“今天”、“明天”和“后天”的所有待办事项都应该是小的任务（几个小时就可以完成的）。比如，制作注册和登录就是一个不好的待办事项，因为要过好长时间你才能在待办事项清单中看到进展。制作注册和登录可以分解成一系列小任务。

我的个人待办事项清单

- [ ] - - - 今天 - - -
- [ ] 注册：制作 HTML 表单
- [ ] 注册：实现 JS 验证
- [ ] 注册：验证电子邮件唯一性
- [ ] 注册：发送验证邮件
- [ ] - - - 明天 - - -
- [ ] 登录：制作登录表单
- [ ] 登录：实现 JS 验证
- [ ] - - - 后天 - - -
- [ ] 登录：制作“忘记密码”表单
- [ ] 登录：制作“忘记密码”邮件
- [ ] 登录：制作“重置密码”表单
- [ ] 登录：制作“重置密码”确认
- [ ] - - - 未来 - - -
- [ ] 未来将要做的事……

添加项目

这样，我们就把制作注册组件拆分成了一个历时三天完成、由十项任务构成的待办清单，井井有条，却不太复杂。

每个待办事项都是一小块工作。一旦我们完成了一件事，就可以把它勾掉。这样我们每完成一件事时，就会立刻有一种满足感。而且也不用等待完成整个组件，随时都可以看到进展。

到一天结束时，我们可能没有完成所有计划的项目，常常会剩下一两项。太阳落山时，我们的待办事项清单常常看起来是下面这个样子，今天还剩下几项没有完成。

我的个人待办事项清单

- [ ] - - - 今天 - - -
- [ ] 注册：发送验证邮件
- [ ] - - - 明天 - - -
- [ ] 登录：制作登录表单
- [ ] 登录：实现 JS 验证
- [ ] - - - 后天 - - -
- [ ] 登录：制作“忘记密码”表单
- [ ] 登录：制作“忘记密码”邮件
- [ ] 登录：制作“重置密码”表单
- [ ] 登录：制作“重置密码”确认
- [ ] - - - 未来 - - -
- [ ] 未来将要做的事……

添加项目

- [x] 注册：验证电子邮件唯一性
- [x] 注册：实现 JS 验证
- [x] 注册：制作 HTML 表单

明天如何变成今天

那到了明天要怎么样呢？昨天要是剩下一两项没做完怎么办？只要几秒钟，就可以更新待办事项清单。在 Ta-da List 里面，只要用

鼠标拖动两下即可。

首先，把明天分隔符拖到后天分隔符上面，这样所有为明天设定的内容现在都归到今天了，我们昨天没有完成的事项还留在今天里。再把后天拖到未来分隔符上面，这样所有为两天之后设定的内容都归到明天了。

我的个人待办事项清单

- [ ] - - - 今天 - - -
- [ ] 注册：发送验证邮件
- [ ] 登录：制作登录表单
- [ ] 登录：实现 JS 验证
- [ ] - - - 明天 - - -
- [ ] 登录：制作“忘记密码”表单
- [ ] 登录：制作“忘记密码”邮件
- [ ] 登录：制作“重置密码”表单
- [ ] 登录：制作“重置密码”确认
- [ ] - - - 后天 - - -
- [ ] - - - 未来 - - -
- [ ] 未来将要做的事……

添加项目

- [x] 注册：验证电子邮件唯一性
- [x] 注册：实现 JS 验证
- [x] 注册：制作 HTML 表单

回到未来

每天都要看一眼越来越长的未来清单。如果你需要在未来两天内完成其中一项，把这个任务拆分成小块，然后把它们安排好。

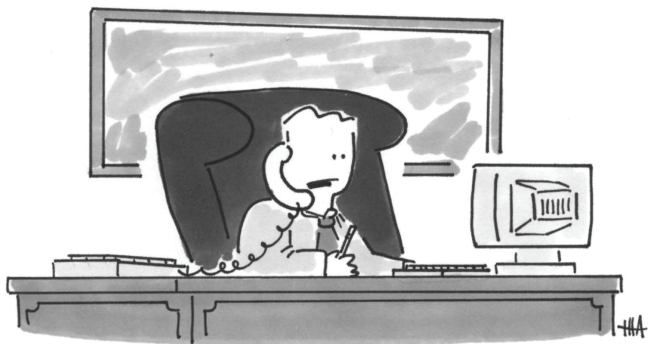
每天重新评估重要事项

个人待办事项清单可以非常完美地调整优先级，它每天随着软件

开发的不确定性而变化。可能有某件事本来是要在今天完成的，但现在看起来不那么重要了，只要把它拖到明天甚至更远就可以了。类似地，本来安排在后天的事项可能你今天有精力做了，那就拖动，完成，然后把它勾掉。

很多时候，我们没法完成今天的所有事项，可能有一个事项卡在今天有好多天了，因为总有其他优先事项挡在它前面。

但过一阵子就会开始浮现出模式。有些待办事项一直在今天一栏中晃悠，或是经常被推迟到明天。这些“坏蛋”待办事项可能并没有我们最初添加时所想的那么重要。如果一个待办事项已经晃悠了一两个礼拜，就直接删掉好了。避免不重要的工作和完成重要的工作一样富有成效。



啊，怪不得！它在我的“不要做”清单上！

个人待办事项清单不是什么黑魔术，它不会替我们把工作做完。不过，它能帮助我们理清头绪并即时做出调整，还能让我们每天都看到真正的进展。我们对每天的小小进展感到欣慰，这让我们明天还能保持干劲。

第23篇 和团队一起安排免打扰时间

你只有在有时间高效工作时，才能有效率。说起来容易做起来难，日常工作中的那些基本活动——开会，打电话，同事交流——实在太普遍了，我们甚至忘了这些实际上都会分散精力。

那其他公司是怎么解决分神问题的呢？

37signals宣传一种远离其员工的哲学。在《重来》（*Rework*）[FH10]一书中，他们谈到，干扰不仅打断了我们手头正在做的事情，还让我们脱离了自然的“独处空间”，就是那种完全专注在工作上的状态。

Google的“20%法则”很有名，它允许工程师每周拿出一天，专门做自己特别喜欢的一个和公司相关的项目。[①]这些额外项目中，很多最后都变成了Google最出名的产品，比如Google新闻和Gmail。

那些希望员工发展而不仅仅是生存的公司，都需要采取一些方法来将其变成现实。

如果可以选择的话，我宁可每天都在家办公。这样就不用担心客户会突然打来电话，或者某人需要我马上前去帮忙，没有了这种烦扰简直是如释重负，我可以成天都专注在代码上。

但我们也要为客户提供服务，有时候这需要花费很多精力。我们很难整天都在编程，总有火要救，有客户要联系，有邮件要回。

免打扰时间，欢迎你

几年前，我在公司里设置了免打扰时间。这是模仿其他一些公司的“别来打岔，让我干活”的规则，同时仍然满足了客户的需求（如

① 参见 http://www.nytimes.com/2007/10/21/jobs/21pre.html。

电话、邮件和一般的关注）。具体做法如下。

每个程序员每天轮流有两个小时的免打扰时间。如果你处于免打扰时间中，那么

- 不用回邮件。
- 不会参加会议。你在这段时间中没空。
- 不接电话。
- 同事不会给你发即时消息。
- 同事不会和你说话。

在免打扰时间里，我们在桌子上竖一个白旗。两个小时之后，我们回过头来查收邮件，回复电话和即时消息，然后照常工作。这条黄金法则很简单：某人处在免打扰时间时，不要去打扰他。

要是全公司一起免打扰就好了，但这对于我们的业务来说不现实。这意味着切断与所有客户的通讯以及内部通讯两小时。对于某些客户来说这个时间太长了，特别是我们正处于发布中间的时候。

我们制定了轮流免打扰机制，例如：

- 第一班　每天早 10 点到中午——Ka Wai，Michael
- 第二班　每天下午 2 点到 4 点——Anthony，Mustafa
- 第三班　每天下午 4 点到 6 点——Tom，Craig

因为时间是错开的，所以任何时候只有一两个人处于免打扰时间。公司不会一下子就切断客户联系，大部分时候都可以联系我们。

别人能帮你

对于我们来说，谁进入免打扰时间不是随便安排的。我们在每个免打扰时间段内，都把不同职务的成员搭配起来。通常，我和 Mustafa，Tom 和 Mike，Anthony 和 Craig 工作联系最密切，所以我们不会同时进入免打扰时间。

这意味着如果有紧急情况，搭档可以处理。如果 Craig 在早上 11:30 要问我一个问题，他可以去问 Mustafa。如果我在下午 4:30 有问题要问 Tom，我也可以问 Michael。如果客户在某人的免打扰时间安排了电话会议，他的搭档就可以在这个时间代替他。

打扰别人是最后选择

由于免打扰时间让我们无法和这个人联系，这会让每个人都更认真地思考自己的问题，我们更可能问自己："这个问题我能不能 Google 一下就解决而不用去打扰别人呢？"打扰别人是迫不得已的办法。

免打扰时间每周为我们每个人创造了没有干扰的十个小时，却几乎没有影响与客户和内部的联系。这是现实中保持生产率的好方法。

所以，无论只是几小时的免打扰时间，还是一整天做自己事情的自由时间，或者是在物理上相互远离，实质上都是认真考虑一下，如何可以互不打扰，让自己更为高效。

第 24 篇　采用自治小团队的工作形式

很多大公司都在兜售“人是我们最宝贵的资产”的陈词滥调，听起来就恶心。这些公司同时在用胸牌编号来标识员工，用发购物卡的方式来挽救员工迅速消退的积极性，结果呢，还不是人员流动率居高不下。不过，这也没啥大不了。

有些公司，特别是非常大的公司，可以不停地招聘，用一批人来补另一批人，以保证公司持续运转。他们可以继续和其他的大公司签订巨额合同，在一年内就把新员工青春洋溢的活力挤得干干净净。一旦榨干了，就又去市场上找人。雇人越多，补人越多，“还过得去”的工作职位也就越多，公司就这样维持着。

我们常常认为这是“大公司”导致的企业现象。繁文缛节，官僚作风，开会无休，优柔寡断，犹豫不决。然而与此同时，他们依然高举那条公司格言：人是我们最宝贵的资产。

说老实话，在我的小咨询公司里，人不是最宝贵的资产。我非常清楚，有（许多）比我更优秀的程序员、设计师、思想家和作家。没有谁是一行中最优秀的，总是人外有人。

但造就我们较高生产力的是随着时间建立起来的工作关系。我已经和几乎同一批志趣相投的人一起工作了许多年，这种团队内部的亲密感意味着我们知道每个人喜欢的工作方式。

有些人喜欢细致周到地工作，每一行代码都深思熟虑；有些人则是粗枝大叶，再事后清理；有些人经常需要独自工作，单枪匹马解决问题；而有些人则一上来就需要协作。慢慢地，我们会以不同的方式相互弥补，我们会适应周围的人。随着时间的推移，我们开始凝聚在一起，真的是“凝”在一起。

与此同时，这种熟悉让我们在辩论的时候无需担心伤害对方的感情。在一条战壕里战斗多年的战友，没必要再去相互试探想法。我们可以就一个自己赞同的想法争得面红耳赤，不用小心翼翼地担心伤害对方的感情。我们开会时会非常地活跃、投入，有时候争论会几近白热化。我们要把问题捋顺，找到解决方案。和普通公司那种充满了问题和模棱两可回答的会议比比吧。在那种会议里，避免冲突似乎比找到最好的答案还要重要。

这也就是为什么我说，程序员最好的工作环境是人员流动率很低的自治小团队。在大公司你也可以找到这样的团队，但小公司里更常见。在如今的环境下，小公司才能经常冒出大作品，因为他们可以更快地决策、研究和学习，并且没有繁文缛节的羁绊，进而可以更快地开发。想想那些被许多人使用的、非常成功的，却由很小的公司开发的软件吧。下面只是其中几个例子。

- Campaign Monitor：给 Web 设计师用的邮件营销软件
- Litmus：邮件预览监控
- GitHub：软件开发项目用的版本控制库
- Braintree：SaaS 服务用的在线支付网关
- Basecamp：项目管理和协作软件
- 愤怒的小鸟：移动益智类视频游戏

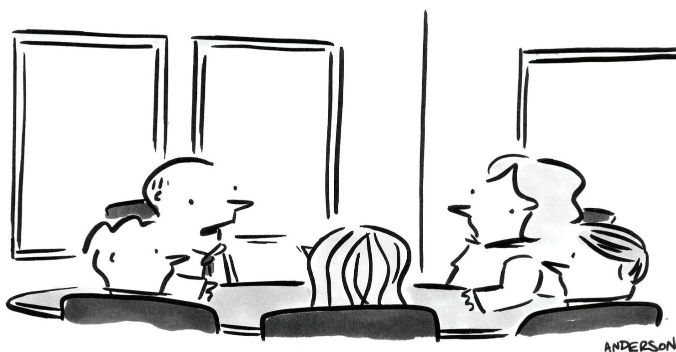
这种相互熟悉性最终把一群优秀的个人变成了一个生产非常出色的产品的伟大团队。

第 25 篇　提高生产力，避谈“我们”

在高效的团队开发中，明确责任很重要。是的，知道谁负责哪一块很关键，但知道谁不对那一块负责也很重要，他们有其他的责任要承担。

当你和同事或客户沟通的时候，特别是开会和写邮件的时候，改掉那种说“我们”的习惯，要明确地说出“谁”。

下次开技术会议的时候，留意一下你讲了多少次“我们”。你说“我们”的时候，真正的意思可能是“我们中的某些人”、“我们中的几个人”，或者很多时候只是你自己（或是 John 或是 Mary）。你的“我们”很少是指全体所有人。



“好，既然我们都同意了，就回到座位上讨论一下为什么这个不行。”

但“我们”这个词给人的印象就是这个样子。“我们”一下子毫无必要地把责任扔到了每个人身上，它不去区分真正需要操心的人和需要专注其他工作的人。更重要的是，它掩藏了需要提供反馈时，你

真正需要联系的人。

比如，你正在写一个工具，用来把遗留数据库中的数据转出来，你需要知道 Customers 表是归哪个 schema，但这问题你不需要问整个公司。前端设计师很可能不知道或者不关心。即使数据库是一个团队在管，也应找出那个真正能给你想要的东西的人。不要说我们需要找到这个 schema，而要说具体谁应该给你提供这个信息。

如果你不知道应该找谁，随便挑一个人。他要么会给你需要的答案，要么会把责任转交给正确的人。明确责任就可以直接把事情向一个方向推动：前方。

这正好和“我们”这种思维模式的结果不同。“我们”这个说法会让 15 个人一起听一通电话，会将同一封邮件抄送给 12 个人，或让一桌子人记录着他们不理解或者对他们不重要的事情。“我们”造成了一大群人参与谈话，而实际上只有少数几个人真的知道在说什么。“我们”让事情的噪声呈指数级上升。

“我们”带来噪声

问题是，“我们”不仅会产生噪声，还会让噪声成倍放大。如果你把“我们”带到了谈话中，就是在让谈话变得越来越拖沓冗长。

为什么？因为“我们”让你问出一些没必要马上回答的问题，这些问题经常像是自问自答。这会不断引发无关紧要的新的对话，偏离了你开始真正想寻求答案的问题。

- “我们”觉得加上这个功能怎么样？
- “我们”需要给 Web 服务器加硬件吗？“我们”能找出性能指标吗？
- “我们”能做什么来改善用户体验？

这些问题经常是在会议要结束的时候提出来的，以便安排另一个会议。即使真正做决策的只有一两个人，每个人也都会说“我们需要再想想”。在小组会议或是邮件中问这些问题很正常，但若把冷冰冰的“我们”换成真正的人——你周围的那些温血动物——这些问题一下子就变得直接了。

- Jennifer 觉得加上这个功能怎么样？
- Mike 需要给 Web 服务器加硬件吗？Anthony 能找出性能指标吗？
- Tom 能做什么来改善用户体验？

你把问题指向某个人的时候，人们就会行动了。

旁观者效应

“我们”类型的问题经常会冒出来，因为这是人类的本性。这和旁观者效应毫无二致：出了紧急状况的人，与向一大群旁观者求助相比，向单个旁观者求助很可能会更有效。

根据社会影响的基本原理，在紧急情况下，旁观者会观察他人的反应，看看他人是不是觉得有必要帮忙。由于每个人都做同样的事情（即什么也不做），所以他们都会从他人的无动于衷中得出结论：不需要帮助。[1]

旁观的人越多，有人采取行动的可能性就越低。所以，要明确指定对任务负责的那个人。

虽然听起来很简单，但把“我们”换成具体的人名（甚至是“我”）会让团队的生产力大大不同。在会议中，在邮件中，或是在电话中，使用“我们”都会削弱动力，让你的团队不清楚谁应该干什么，以及

① 参见 http://en.wikipedia.org/wiki/Bystander_effect。

谁不应该干什么。“我们”对于生产力来说就是高果糖的玉米糖浆[①]，听起来很甜，但里面满是有害健康的东西。

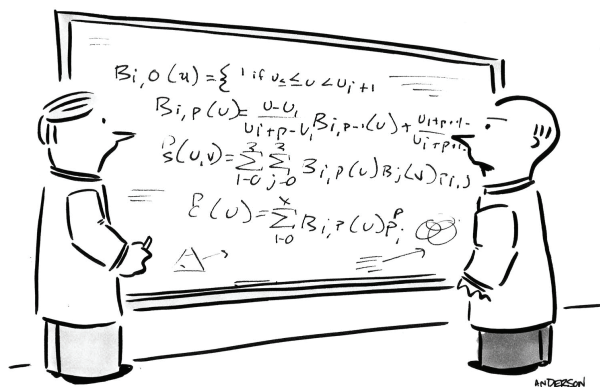
我们在上面几篇文章中已经看到，有时候提高生产力只需降低复杂度：和更少的人工作，将责任交给一个人，或是消除团队开发中的外部噪声。在下一章中，我们会关注软件中的复杂度。

① 有研究称，高果糖玉米糖浆可能会引起肥胖、心血管疾病、糖尿病、非酒精性脂肪肝等健康问题，但仍有争议。——译者注

第 5 章

复 杂 性

除了死亡和税收[①]，复杂性可能是生活中唯一肯定存在的东西，它总是随着时间的推移而增长。在我们这个行业，复杂性是软件日渐成熟不可避免的后果。除非我们愿意删除软件中的功能，否则根本没有办法避免。



“这到底是什么意思？！”

① 出自富兰克林的名言：“在这个世界上，除了死亡和税收，没有什么是确定无疑的。”——译者注

如果我们不能摆脱复杂性，那么下一步就只能抑制它的增长。我们需要认识到什么时候复杂性没有必要，还要培养灵敏的“嗅觉”来寻找它。如果我们知道它是什么“味儿”，并且可以在软件的每一个角落找到那再熟悉不过的恶臭气味，我们最后就能得到更好的结果。

第 26 篇　“嗅”出坏的复杂性

所谓坏的复杂性，就是根本毫无必要的复杂性。

这并不总是很容易发现。有时候，即使我们认为某样东西很有必要，但实际上也并不是这么回事。这种情况甚至曾发生在托马斯 · 杰斐逊身上。

1776 年，杰斐逊是起草《独立宣言》的委员会成员，并有幸撰写第一稿。草稿完成后，他把稿子寄给他的朋友本杰明 · 富兰克林审阅。富兰克林回信的时候，把杰斐逊许多精妙的语言都删除了。

杰斐逊不太高兴，但富兰克林试图说服他的朋友为什么这样更好。富兰克林给他讲了这个故事：

> 当我还是印刷厂学徒的时候，有个伙伴是制帽学徒，他已经满师，打算自己开店。
>
> 他首先想到的是要有一个漂亮的招牌和恰当的题词。他写了这么一些话。
>
> “约翰 · 汤普森，制帽师，制作和销售帽子换取现金”
>
> 后面还画上了一顶帽子。不过他觉得应该发给朋友们，请他们帮忙修改一下。
>
> 第一个看到的人认为“制帽师”多余，因为后面紧跟着“制作帽子”，这已表明他是一名制帽师。于是这个词被剔掉了。
>
> 第二个人觉得“制作”也可以省略，因为客户不会关心帽子是谁做的。如果质量好又合意，他们就会买，不管它是谁做的。于是这个词也去掉了。
>
> 第三个人觉得“换取现金”也没用，因为当地并没有赊

> 账的习惯，每个人买东西都要付现金。于是这个短语也去掉了，现在的题词就变成了“约翰·汤普森销售帽子”。
>
> “销售帽子！”下一个朋友说，“可没人会希望你免费送啊。你写这个有什么用？”于是“销售”也被去掉了。“帽子”也跟着被去掉了，因为招牌上已经有一幅“帽子”的画了。

最后，招牌就被精简为“约翰·汤普森”，后面画了一顶帽子。①

很多软件都可以从这个制帽师的故事里得到一点启示。这个按钮非得有吗？复制这一行增加了任何价值吗，还是只是重复了已有的东西？这项新功能真的有助于任务的实现吗？

考虑一下，你可以从软件里面去掉什么，却丝毫不会影响功能。

① 参见 http://www.pbs.org/benfranklin/l3_citizen_founding.html。

第 27 篇　关于“简单”的悖论

让复杂性成为一个奇怪现象的原因是：每个人都喜欢简单。所以人们常说“我只是想让事情简单一点儿”。有谁说过“我只是想让事情复杂一点儿”？

我决定找找答案，于是我在 Google 上搜索了一下。

在写这篇文章的时候，如果你 Google“我想让事情简单一点儿”，大约会有 954 000 条匹配结果，而“我想让事情复杂一点儿”只有唯一一个匹配结果。

一个。那么唯一的一个匹配结果是什么？那是我于 2009 年 10 月就此话题写的一篇博客文章。如果把我自己从人类文明的记录中剔除掉，那么很显然，没有人想要甚至想过主动把事情变复杂。

那么，为什么我们自己做东西的时候就遇到了杰斐逊一样的复杂性问题呢？为什么我们生产的东西常常复杂无比呢？很多想法很好的软件怎么就从简单的点子变成了复杂功能的恶梦呢？

简单的产品实际上不好做

大多数想法，表面看起来简单，可着手细节的时候却极度复杂。高质量的想法总是很简单。每一个商业构想都必须能进行电梯游说，即能在 60 秒内把想法从头到尾讲清楚。我们没法在 60 秒内把复杂的东西讲述清楚。

当觉得想法开始变得复杂的时候，我们就离开了舒适的“想法”王国，进入了赤裸裸的“实施”的现实。一旦我们深入细节，就会发现所有的逻辑漏洞都在这里。这就是细节的本质。到了这个时候，（大部分）没有经过充分考虑的想法都不太可能不变复杂。有时候，低着

头蒙着眼干活比重新思考这个想法要容易，于是就做出了一些草率的决定，为了坚持神圣的“伟大想法”而添加了功能。然后，软件就变得越来越复杂了。

简单有时似乎不够

如果每个人都喜欢简单的软件，而大多数软件做起来不简单，那似乎最好的情况就是软件既简单易用又容易制作。这对用户和开发人员来讲是双赢，但世界上很少有这种软件。这个谜里面还涉及更多的东西。

答案就在于我们自己对不足的恐惧。当我们做了一个简单的东西时，就感觉它……不够复杂。我们说服自己来相信，客户的付出并没有得到相应的回报。一个本身很简单、做起来也简单的东西让人感觉毫无价值，一个很容易实现的想法也很少被视为“伟大的想法”。

风险投资人不会为简单的想法倾注数百万美元，他们把所有的钱都扔在唐纳德·特朗普式的极端追求上[①]。这是同类中最好的吗？有创新吗？是最尖端的吗？很多情况下，这无非是换了种方式来说“一个想法足够复杂，所以很有价值”。

矛盾就在于此。作为开发者，我们往往把所做软件的价值等同于其复杂性，复杂度越高就等于价值越大。

用户则有不同的看法。现实情况是，90%的用户只使用普通企业级软件中 10%的功能。如果用户无法在众多不需要的功能下面找出需要的那个，他们要么怪自己能力有所不足，要么就归咎于软件本身。开发者和利益相关者把简单看做缺点，而用户却把复杂看做缺点。

---

① Donald Trump，美国商业大亨、电视名人、畅销书作家。他制作并主持著名的电视节目《学徒》(*Apprentice*)，每周至少淘汰一名候选人，最终聘用一名员工。
——译者注

在我的公司，我们一直抵制把事情搞复杂的自然欲望。我们必须时时提醒自己，要把软件做得简单一些。

我们就自己软件的功能进行过无数次的内部争论，但最后都采用了简单的解决方案。有时用户界面只需要对文本做出小的调整，有时就是重新整理一下链接。有时候我们会就一个新功能争论几个小时，最终决定不值得为了这个功能增加软件的复杂性。

我们要学到的教训是，并非就功能讨论了好几个小时，就一定要添加好多个功能。我们很自然会觉得，花在一件东西上的时间应该和其可度量的产出相匹配，而不管新功能到底能带来什么好处，但是请摆脱这种折磨人的想法吧。一旦你不再天真地认为简单会降低价值，就可以开发出好的软件了。

一个简单的解决方案不应该被认为是“缺了什么”，有时候，它恰到好处。

第28篇 复杂性就像挑棍游戏

挑棍是一种古老的儿童游戏，参与者要把一堆塑料棍一根一根挑出来，挑的时候还不能碰到其他的小棍。开始的时候把一大把小棍抓在手里，然后撒开，之后大多数小棍会在中间堆成一堆，少数则会滚到一边。

游戏的目标是要从整堆中挑走尽可能多的小棍，每次一根，不能碰到其他的小棍。如果碰到了的话，你的这一轮就算输了。

维护复杂的软件有时候感觉很像这个游戏，其中每个小棍都代表一个特性或功能。有时候一个特性可以完全孤立，与其他特性毫不相干；有时候它会影响几个组件；还有些时候，它和许多其他特性紧紧地交织在一起。

植入新功能就像在堆里再加上几根新的小棍。到了某个时候，抽走任何一根小棍而不影响其他的小棍都几乎是不可能的了。复杂程度急剧增加。

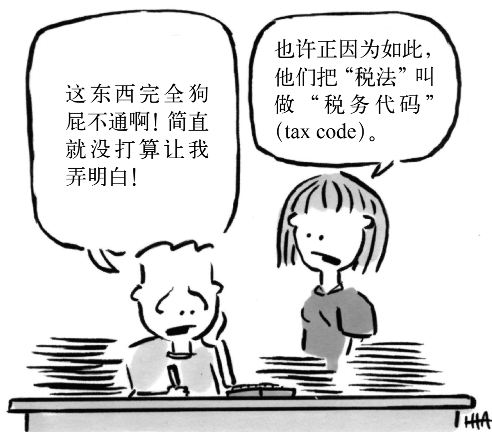
作为开发人员，我们通过养成下面这些好习惯来尽可能避免这个问题：封装代码、将变量定义到合适的作用域、把大块逻辑拆成小块，或者引入模式。我们努力让所有的小棍都并排摆放、互不接触，但在不断加入新的小棍的同时，要坚持把代码重构到“正确”的位置实在有点棘手。这个防线很容易就垮掉了。

每次加入新特性的时候，我们都可能会干扰其他一些最初看起来并不直接相关的特性。加入的特性越多，干扰之处也会迅速增多。

第 29 篇　把复杂性藏起来

“极其复杂的一团乱麻。”

美国国税局的全国纳税人权益维护人 Nina Olson 是这样形容美国税务系统的。美国联邦个人所得税法案差不多有 6500 页。你能想象吗？你觉得我在撒谎？那你说对了。它事实上有 65 000 页。



这就是我热爱 TurboTax 的原因，它是美国最流行的所得税报税软件之一。它把这 65 000 页文档拿过来，然后奇迹般地弄出了某种像我这种普通人都能够使用的软件。TurboTax 本来可以只是把联邦和州税的 1040 表[①]数字化，把每个项目变成一个文本框，然后再做上一点点计算，最后通过信息高速公路把表格提交。

也许那样它仍然会有许多粉丝。

① 1040 表是每年个人申报收入时需要填写的表格，对于联邦税和州税各需填写一份，如有较为复杂的收入和支出情况还需要另附其他表格。——译者注

但是，TurboTax 没有这样做。它就像一个个人税务向导，明白没有人真的愿意花心思琢磨它。“把你的 W-2 表格拿出来[①]，把数字抄给我。”“你有农场吗？没有？！那我们就再也不用提这个了。”它甚至知道，在它问我一些古怪问题的时候，要告诉我这个情况十分罕见——“这对你也许并不适用。”

TurboTax 真是一个英雄，它让报税成为至少可以接受的一件事。我都能想象这背后的逻辑是多么地复杂。不光是要有联邦法的内容，五十个州和数以千计的县都要有各自的法案。对于每一个地区来说，单身的、结婚的、小型企业主、投资者、学生、慈善家、首次购房的、失业的、刚退休的、常忘事的、穷人、富人、富豪，啊对了，还有农民，都得有不同的程序。

再加上每年税法都会有所变化。为了政府开支中的某个专项资金所去掉或者加上的每个小规则，最终累积成了这 65 000 页的文档。如果你为多欠了政府几百美元而心怀怨怼，想想那个写 TurboTax 的程序员吧，他还得为过去六个月中在密西西比州买了个环保摩托艇的家伙再多写上一点儿诡异的条件逻辑呢。

那这些代码有没有重构的价值呢？如果今天写的这段代码到下次国会休会的时候就过时了，或者将来依赖的参数完全不同了，他们还能重构什么呢？

TurboTax 证明了，即使手头的任务极度复杂混乱，软件也大可不必如此。你可以把所有的复杂性都隐藏起来，把这团乱麻转化成可以让人理解、让人使用的功能，从而创造出真正有意义且非常简洁的软件。

---

① W-2 表格是全职雇佣关系中，雇主在年末发给员工的全年工资和所得税预扣情况的清单。——译者注

第 30 篇 “难编”可能意味着“难用”

TurboTax 把一个几乎不可能理解的东西变得易于众人使用，它把复杂性从用户那里转移到了代码里。

然而，这种转移并不总是一个零和博弈。有时候，过分复杂的逻辑是软件功能模糊不清的信号。用复杂代码去支持复杂界面？我们能不能不要这些呢？

电梯中的困惑

想像我们在一个工程师小组里，试图为一栋 50 层高的大厦设计一套更好的电梯控制软件。电梯可以监控人们从哪层楼进入以及按了哪层楼的按钮。我们的经理走进来，只提出一条要求：人们都抱怨在电梯里面等待的时间太长，设计的时候要让人们呆在电梯里的总时间最短。



“不，我肯定是错过去了。”

我们开始集体讨论各种场景。假设约翰从一楼进来，按下按钮要到他位于第 50 层的顶层公寓。半途中，电梯停在了 8 层。UPS 的快递员史蒂夫走进来，拿着一个棕色大纸箱。他要给 5 层的人送包裹，于是他迅速地按了 5。

那电梯应该怎么办呢？是因为离 5 层比较近而应该先下到 5 层，还是因为正在向上走而应该后停在 5 层呢？

如果电梯下到 5 层，那么约翰和史蒂夫总共要走过 58 个楼层。约翰先向上 7 层，向下 3 层，然后再向上走完剩下的 45 层，到达他的顶层公寓。史蒂夫只需要向下 3 层。

- 约翰：向上 7 层 + 向下 3 层 + 向上 45 层 = 经过了 55 层。
- 史蒂夫：向下 3 层 = 经过了 3 层。
- 约翰和史蒂夫总共经过的楼层数：58。

我们来比较一下另一种方式。如果电梯继续向上走，然后再下到 5 层，那么他们就要经过两倍多的楼层数。

- 约翰：向上 49 层 = 经过了 49 层。
- 史蒂夫：向上 42 层 + 向下 45 层 = 经过了 87 层。
- 约翰和史蒂夫总共经过的楼层数：136。

答案很明显。电梯应该向下，先送史蒂夫和他的棕色大纸箱，然后再一路上到约翰的顶层公寓。我们都觉得新电梯肯定会棒极了！

我们再继续想象一下。假设 UPS 快递员史蒂夫是在 30 层而不是 8 层进来的，那么电梯要是先下来就会出现下面这种结果。

- 约翰：向上 29 层 + 向下 25 层 + 向上 45 层 = 经过了 99 层。
- 史蒂夫：向下 25 层 = 经过了 25 层。
- 约翰和史蒂夫总共经过的楼层数：124。

那电梯要是先一路向上，到约翰的顶层公寓之后再向下呢？

- 约翰：向上 49 层 = 经过了 49 层。

- 史蒂夫：向上 20 层 + 向下 45 层 = 经过了 65 层。
- 约翰和史蒂夫总共经过的楼层数：114。

在这种情况下，如果电梯向上，我们就节约了 10 个“人层”！所以，根据史蒂夫从哪一层进来，电梯会决定是继续向上，还是先向下再向上。如果是两个人乘电梯，那么编程解决他们乘电梯的最优方案还算是个简单的任务。

现在假设还有第三个人——萨曼莎进入了电梯，那么我们让谁先下电梯就有了六种可能的方案。我们的程序需要计算每种方案所经过的总楼层数，然后找出最优路线。

- 情况 1：约翰，史蒂夫，萨曼莎
- 情况 2：约翰，萨曼莎，史蒂夫
- 情况 3：史蒂夫，约翰，萨曼莎
- 情况 4：史蒂夫，萨曼莎，约翰
- 情况 5：萨曼莎，史蒂夫，约翰
- 情况 6：萨曼莎，约翰，史蒂夫

事实上，需要测试的情况数就是任意时刻电梯中总人数的阶乘：

- 2 个人 = 2! = 2 次比较
- 3 个人 = 3! = 6 次比较
- 4 个人 = 4! = 24 次比较
- 8 个人 = 8! = 40 320 次比较

一旦数目超过了几个人，要测试所有情况就变得不可行了。但这只是复杂性问题的一个方面而已。

人们会在不同的时间上下电梯。每次有新的人进入电梯，我们就需要记录现有的乘客已经经过了多少楼层，然后再开始新的计算。

换句话说，光是看看现在电梯里面有谁，我们无法推断出电梯已经走过的路线。如果迈克在 25 层进来，然后桑杰在 35 层进来，那么

在迈克和桑杰先后进入的这段时间里，电梯是只走过了 10 层，还是先下到 21 层去送了萨曼莎呢？

有两位以上乘客的时候，我们还必须记录人们什么时候下了电梯，这样之后的计算中就不用考虑他们了。

此外，要是有人忘了按楼层，或是按错了楼层，然后半路上又按了个按钮呢？我们的软件需要重新计算，并且可能要半路改变方式吗？

如果你是编程新手，很不幸，一点都不夸张，诸如让人们尽快走下电梯这样一个看似简单的目标，最后可能会变得非常复杂。

没有什么回报的复杂性

假设，克服了那些额外的障碍之后，我们设法做出了完美的系统，写出的代码可以让大家呆在电梯里的总时间最短——在一瞬间就计算了成千上万的情况，并考虑了每个人走过的路线。这真是一座技术的丰碑！但约翰、史蒂夫和其他人感觉如何呢？可能并不怎么好。

代码最擅长处理那些繁冗又可以自动化的东西，诚然这就是一个极端繁杂的例子。但人却不太擅长这个。站在电梯里，没人能够很快地在脑子里计算所有的情形。等着走下电梯的人完全是受电梯摆布，不知道下一个人上来以后会往哪个方向走，也不知道为什么。

这就是一个*难编就意味着难用*的例子。在这个例子里，复杂性对*双方*都造成了损害。只要有第四个人走进电梯，情况就会变得太复杂，人脑就无法知道电梯想往哪个方向走了。即使电梯正在实现让所有人走过的总楼层数最少的目标，电梯里的人也不知道什么时候才能轮到自己下电梯。

他们可能更喜欢那种简单传统的电梯算法。

当然，这可能并不是最优线路，但采用一种比较简单的方案，电

梯里的人就更了解现实的情况。用户对可预见性的要求胜过了更复杂的方案，即使这种方案在理论上效率更高。

如果细节变得异常难编，这可能意味着系统的实际功能难以让人理解。你可能在编好了无比复杂的东西之后松了一口气，但别人在用过之后可能对你恨得咬牙切齿。

第 31 篇　知道何时重构

写代码时要是想得太长远，也会导致复杂性的问题。对要编写的实际的东西太过敏锐或是太过理性，是要付出代价的。一个经典案例就是过早地运用设计模式。

不要误解我的话，设计模式是很好的东西。一个通用的编程方法反复出现之后，我们就会兴奋起来。我们都体验过这种感觉：自己的代码不仅可以完成手头的具体任务，还可以做更大的事儿。

有了几次这种体会，成功地把代码重构成更抽象的模式之后，我们很容易有种所向披靡的感觉。我们像警犬一样，嗅出所有的小信号、小线索，以及所有可以对我们直白的代码进行抽象的暗示。

但很快，我们的第六感就会反咬我们一口，让我们鲜血淋漓。很多人都听说过或者体验过“用架构把自己逼入死角”的恐怖故事，这就是用抽象方法来解决问题时用得过头了。

过早重构的危险

打个比方，假设你正在为“萌芽网络”公司做内网，该公司不大，只有两个部门：IT 和销售。公司的高管希望能够根据一组员工参数来计算员工的预期奖金，但每个部门计算奖金的标准却不同。

IT 部门只想按照员工当前工资的某个百分比来发放奖金，并且只发给在公司工作了五年以上的员工。销售部门希望给所有人 1000 美元的基础奖金，再根据他们在公司工作的年限，每工作一年就增加 500 美元的额外奖金。毕竟，公司正在蓬勃发展，高管也都挺慷慨的。

于是你开始编程。你先定义了一个 Employee 类，包含计算员工奖金所需要的所有信息，然后写了个简单的函数，暂时只包含一个简

单的条件语句来返回某个员工的预期奖金。

```
public decimal GetBonusForEmployee(Employee employee)
{
  if (employee.department == Departments.IT)
  {
    //按 IT 部门的方式计算奖金
    if (employee.Years >= 5)
    {
      return .1 * employee.Salary;
    }

    return 0;
  }
  else
  {
    //按销售部门的方式计算奖金
    return 1000 + 500 * employee.Years;
  }
}
```

你写了个小工具来把所有的员工都加载到一组 Employee 对象中，然后对每一个对象应用上述方法。搞定，该去喝一杯了。

但是你开始思考，当萌芽网络公司真的开始“发芽”的时候会出现的其他可能性。要是出现第三个、第四个部门呢？把条件逻辑换成 switch 语句！那还等什么呢？现在就未雨绸缪，为未来做好准备：

```
public decimal getBonusForEmployee(Employee employee)
{
  switch(employee.department)
  {
    case Departments.IT:

      //按 IT 部门的方式计算奖金
      if (employee.Years >= 5)
      {
        return .1 * employee.Salary;
```

```
        }

        return 0;

      case Departments.SALES:

        //按销售部门的方式计算奖金
        return 1000 + 500 * employee.Years;
    }
  }
```

干得漂亮！switch 语句是一个很安全的预见性做法。它明确地写出了“销售部门”，而不是把它放在 else 语句里。如果以后有了市场部，你就知道应该把它放在哪里。

这个小的重构挺有道理的。你的代码现在更明确，也更容易快速浏览，换一个程序员来也可以马上上手。

你意识到了自己更崇高的使命，于是决定继续下去。要是两个部门变成了……十个呢？几个月以后，可能就会冒出很多新部门，比如法务部啊，生产部啊，财务部啊，还有清洁服务部，等等。switch 语句最终会变得很臃肿，它会沾上很多并没有什么业务内涵的复杂运算，它们全都赤裸裸地暴露在这个计算奖金的方法表面。

你急匆匆地翻了翻心爱的设计模式书（我极力推荐 Joshua Kerievsky 的《重构与模式》[Ker04]），找到了策略（Strategy）模式！把所有单个的奖金计算都放到各自的策略类里（比如 ITBonusCalculationStrategy 和 SalesBonusCalculationStrategy），它们都是奖金计算策略接口（IBonusCalculationStrategy）的实现。这个接口要求每个实现类都定义一个 CalculateBonus()方法。

完成之后，你修改了 Employee 类，让它包含一个具体的策略实例，然后建立了一个新的公共方法来返回员工的奖金。

有了策略模式，你现在就可以把 getBonusForEmployee()方法

整个去掉了。对员工奖金的计算可以完全由类本身完成了。于是，所有那些复杂的算法都会优雅舒适地藏在 `IBonusCalculationStrategy` 接口的各个实现之下。

既然都做了这么多了，你决定再来“打扮”一下你的代码。你把员工的创建抽象成了一个工厂（Factory）模式，这样你就可以针对部门来编写员工创建器类，为员工分配相应的奖金策略。

你已经完全去掉了针对部门的条件判断（这由员工创建器类负责了），繁复的计算逻辑也没有了（藏在了各个部门的策略类下面）。这真是太棒了！

一旦有了第 15 个部门，这个架构简直就是一幅宏伟的画卷。

日复一日，月复一月。寒冬已至，萌芽网络公司开始举步维艰。公司没有设立新的部门，并且奖金的逻辑也变了。你重新上手，却在想你那曾经简单的逻辑哪里去了。啊，对了，已经都策略化、工厂化了。

又过了几个月。萌芽网络公司打电话过来，说他们解雇了整个销售团队，只剩下 CEO 和开发团队了。CEO 想保持开发人员的士气，所以仍然想给员工发奖金，但现在完全是看工作年限了。

现在真该哭了。你把所有的赌注都押在了按部门而设的奖金规则上。话说回来，一年之前这还是个挺安全的想法。你大动干戈，大张旗鼓，使之能够解决接下来一百年所有部门的奖金问题，但现在整个重构都付诸东流了。这不光是过度架构造成的混乱，也是变质造成的混乱。你把整个策略类推翻，去掉了工厂方法，并乖乖地决定，现在一个可能令人厌恶但还不错的条件语句就足够了。

模式是很棒的概念，但实施起来却要万分小心。预见未来的逻辑经常会导致意想不到的复杂性。

要说有什么黄金法则的话，那就是不应该强行把软件变成一个或

一组记录完好的设计模式。相反，一个（或一组）设计模式的实施，应该刚好能够满足软件的预期任务以及近期最可能出现的应用场景。

当你学习设计模式的时候，要把它理解为解决特定问题的常规方法，而不是某个问题的固定标准答案。模式都有优缺点。虽然模式会让某些任务完成得更为优雅，但你总会付出些别的代价。既然如今的网络应用都在根据新的客户需求不断变化，想要一开始就找到“完美”的模式是不现实的。

这是说我们一点都不要预见变化吗？不是的。当我们没有注意架构的方向是否正确时，问题就来了。

疏于管理的遗留代码让人头疼

随便挑一个即使必要时编者也不愿意做简单重构的代码吧，你很快就会发现问题的。变量定义的范围不对，或者更糟糕的是，都是全局变量，然后用一些诡异的命名法则来保证它们的唯一性。条件逻辑读起来就像是软件的使用条款：一大堆毫不相干的事实用 and 和 or 硬扯到一起，下一个非得修改它的那个不幸的家伙根本看不出这一团乱麻是什么。

在那些在程序员手中辗转流传的遗留代码中经常能看到这种问题，这帮人上手的时候就没什么激情，完工的时候就更是懈怠。方法识别标志杂乱无章，方法调用乱七八糟，简直连代码自己都不知道自己在干什么，例如：

```
calculateBonusesForTeam(.02, 155000, null, 0, 0, null,
new Employee(), null, null, false, true);
```

随着时间推移，被忽视的重构会让你付出很大的代价。毫无征兆地，代码维护起来越来越慢。不要说大的改动了，即使是小的改动，那些我们觉得理所当然的小改动，都可能让一套长期没有遵循哪怕一

点点基本的良好习惯的代码崩塌。

我们回到萌芽网络公司的例子上。某个时候，将奖金计算重构成策略模式可能会有意义，把员工的创建方法变成工厂类也可能会有用，只是当时的时机不对。

在开发周期中过早地过度架构，就会留下一个没填满的坑，而架构不足，就会让我们丧失继续改进软件的选择或动力。

> “漏洞和补丁应当大小相当。”
>
> ——出自“托马斯·杰斐逊致詹姆斯·麦迪逊”

也许杰斐逊只是把他和本杰明·富兰克林在几年前的对话传达了一下。

有所预见，但要谨慎预见。无论是小的改动，还是大的模式变化，每次决定重构的时候，都要知道你会得到什么、失去什么。

第 32 篇　确定编程的节奏

那我们怎么处理预见过早和反应过迟的问题呢？

把软件开发想象成开手动挡的车吧。起步的时候，我们挂的是一挡：稳稳地迈出编程的脚步。编得越多，我们的效率就越低。到了某个时刻，我们就得加挡了。

加一挡，用编程的话来说，就是清理代码：退后一步，重构、抽象或是实施一种模式。这意味着，在开发过程中，到了某个时候，要花些时间来考虑如何改变我们的习惯。这样做并不意味着我们先前做错了。相反，这是自然的，是必要的。

编程和开车一样也要换挡：要是换得太早，就得花好多时间才能恢复到原来的速度；要是换得太晚（或者干脆不换），我们的代码就会进展缓慢。所以，知道什么时候换挡是很重要的。这会让开发过程尽可能高效地运行。我们不是为了换挡而换挡，而是在该换的时候必须换。我们要找到自己的编程节奏。



“我开得太快了，所以我觉得我还是盯着路吧，不要看速度表了。”

软件开发中没有一定的挡数。编程的节奏里可以只有 5 挡，也可以有 500 挡，这取决于项目的复杂度和规模，以及我们自己换挡的意愿。越复杂的项目，就要多留出些挡位，这样我们可以换得更勤。这意味着如果我们换得早了点儿，用不了多少时间就可以恢复速度；要是晚了点儿，也不会对进度产生太大影响。对于小项目来说，只要几个挡就够了。

归根结底，软件的复杂性是必要的，这是为添加更多的功能所付出的代价。关键是要知道什么时候复杂是好的、什么时候是不好的。让自己的第六感告诉你，这一次，复杂性让所有人都受害。正是因为这个原因，我们的工作更像是艺术而不是科学。

复杂性是我们在行业里浸润多年之后才能够更好地理解的东西之一。编程老手都能够很好地管理它。总体来说，我们需要把这种智慧传承给未来一代代激情洋溢的程序员们。在下一章里，我们会谈谈如何成为更优秀的程序员，同时又成为更出色的老师。

第 6 章

教　　学

不论在什么领域，教学都是最难掌握的事情之一。世界上有许许多多的专家，但是教育家就少得多了。

教学不光是把我们知道的东西说出来，更是一门艺术，它需要我们将自己的所学传授给一个正在学习新知识的人。掌握知识只是成为一名成功的教师所必备的要素之一。

我们经常会希望一个学识渊博的人也是一位优秀的教师，但事实并非总是如此。以赛亚 · 托马斯是跻身 NBA 名人堂的著名控球后卫，他 12 次入选 NBA 全明星赛，并率领底特律活塞队两次夺取 NBA 总冠军。然而，作为一名主教练，他的战绩平平——187 胜，223 负。有些人拥有渊博的学识和高超的技术，却无法将这些特质传递给他们的学生。

给新手讲授编程概念可能还要困难得多。我们没有类似于胜负场数这种明显的指标，很难衡量学生是否真的接受了讲授的内容。

对于我们来说，从发现问题到解决问题的过程可能相当复杂，很难描述我们究竟是如何找到最后的解决办法的。但描述清楚这个过程至关重要，因为这是我们培养新一代激情洋溢的程序员的最好方式。

在这一章里，我们会讨论一些需要避免的常见行为，以及如何帮助他人掌握我们头脑中所有复杂的知识。

第 33 篇　教学不同于编程

乍一想，出色的程序员似乎应该拥有成为优秀教师所需的全部技能。

毕竟，编程和教学还是有很多共通之处的。在那些花里胡哨的语法下面，代码不过是一套实实在在的指令集，告诉框架如何去做某些事情。即使有一个极其微小的细节落下了，我们也能够很快发现，因为编译器会“大声告诉”我们的。

编程也要求我们按照一定的顺序来进行。我们不能还没定义概念就去实现它，正如我们不能在一个人还不明白加法的时候就教他乘法。

另一方面，编程和教学可以说是相去甚远。实际上，编程的一些坏习惯完全无益于教学。

首先，编程很少是按部就班进行的。我们写代码时不会一页纸从头写到尾，而是不停地在代码里跳来跳去，旁观者可能看不出我们实现功能的顺序。这在我们调整细节的时候特别明显——这里改个变量名，那里修正一个数据类型。要是按照编程这一套来搞教学，那讲课的人就会结结巴巴、反反复复，而听课的人则会云里雾里、迷惑不解。

其次，编程可以让我们暂时不考虑细节。例如，在重新定义一个公用方法的输入参数时，我把方法的识别标志改掉了，然后重新编译。我明知道这个代码是编译不过去的。我知道会弹出一串错误，只要一调用这个方法，它们就会跳出来说参数类型不匹配。但我这样做的原因在于，比起看遍所有的代码或是手动查找调用，这种方法能更加快速地找出我需要修改的其他地方。

通常，我们编译代码并不是因为觉得工作已经完成了，而是想要

找出哪里还有缺失。编译器是懒惰的程序员最好的朋友，单元测试、代码提示和自动补全也是。

所有这些细节对于高效编程来说都是至关重要的。它们就像柔软而有弹性的墙，把我们的代码“弹”回来。它们使我们得以先注重大局，而不用太关注代码是否完美。好的编程平台会自动帮我们“擦净嘴角”、修正语法，并且在我们写出的指令不那么连贯的时候，告诉我们真正的意图是什么。编程越快、越高效，我们就越依赖于这些工具来找到正确的方向。

教一个新手可完全不是这么回事。忽略的细节就全都丢掉了。我们不能说前半句，然后指望学生接上后半句，至少一开始不行。而且，人与编译器不同，一旦我们改掉错误，编译器就会把过去的错误忘得一干二净，但人无法将对与错分得清清楚楚。我们可能要编译十几遍才能让代码跑通，但是想想看，要是教学的时候也反复修改多遍才搞对，那学生该多么不知所措。

高效编程的这些古怪习惯实在是和循序渐进的教学原则格格不入。你不能把一堆概念扔给学生，省掉几个细节，然后希望他们能够准确领会你的意思。你不会得到错误代码和警告，却只会看到学生们茫然的眼神。

就算是无比杰出的程序员，也不一定是合格的教师。

第 34 篇　当心“知识魔咒”

Chip 和 Dan Heath 兄弟合著了一本畅销书《粘住：为什么我们记住了这些，忘掉了那些？》（*Made to Stick: Why Some Ideas Survive and Others Die*）[HH07]，作者称一旦你成了某个领域的专家，就几乎不可能明白对这个领域一无所知是什么感觉。

想想你该如何向一个先天失明的人解释什么是颜色，或者向先天失聪的人解释什么是声音。举个不那么极端的例子，想想律师吧，如果没有各种抽象和限定，他就没法准确回答法律问题。

他们把这种现象称为“知识魔咒”。



“我以为我说‘天啊，你们都在说些什么啊？’的时候是代表所有人。”

毫无疑问，栽在这个魔咒上的正是我们自己。

想象一下，向从来没有接触过 HTML 或任何标记语言的人解释 HTML 是怎么回事。我们会先介绍基本的标签，比如`<p>`、`<br>`和`<strong>`，接下来解释标签里的内容如何继承标签的属性，然后说明每个标签必须以相同的标签里加上一条斜线来结束。

在大家都频频点头之后，我们会看一段简单的 HTML：

```
<p>
  Hello world!
  <br />
  It's a <strong>beautiful day</strong>!
</p>
```

对我们来说，这实在没什么好解释的。这就是一小段话，中间断个行，然后有几个词被加粗嘛。赞！下面我们继续讲解 CSS 和浏览器测试！可能在吃午饭前还能讲讲 jQuery 选择器呢！

从刚刚接触标记语言的人的角度考虑一下吧。他们可能会想到这些问题：

- 为什么我们把文字放在<strong>标签行里，而不把<p>放在<strong>标签行里呢？
- 为什么
标签的斜杠在最后而不是在开头呢？结束它的对应标签在哪里呢？
- 能在<strong>标签里面再加标签吗？要是把 day 这个单词包在<strong>里面会怎么样呢？会让它变得……更粗吗？

我们给学生看的这个 HTML 例子里面充满了许多小小的假定，根本没有想到这些会是问题。对于新手来说，每个细微差别都要解释清楚。不要有假定，即使是把一些标签写在行内而给另一些标签换行是为了代码可读性这个事实，也不能认为是理所当然的。

所以，当你教新手的时候，要把速度放慢一倍。每走一步，都要考虑一下你不自觉做的那些假定，并把这些“显然”的东西作为知识点来解释。经常问问你的学生，他们是不是能跟得上。

当意识到“知识魔咒”的重大影响之后，你就更能捕捉到那些可能被学生遗漏的细微之处。

第 35 篇　用浅显的例子

对于新手来讲，好的例子里是没有抽象的东西的。例子就应该是具体的，能够清晰地——哪怕清晰得过分了——传达想要讲授的东西，而且能够提供良好的语境。

相反，坏的例子里满是抽象的概念和模糊的区分。下面就是一个经典案例。

当清晰遇上Sally

想象一下，我们给刚入门的程序员讲面向对象编程的基础知识。我们也许很自然地就从讨论类的构造函数和对象实例化开始。谈到某个地方的时候，我们随手写这么一行代码：

```
Object myObject = new Object();
```

对于我们来说，这就是一行很无聊的示例代码而已。它说的是要为类型为 `Object` 的对象创建一个实例，叫做 `myObject`。我们都知道，`myObject` 只不过是给这个新创建的实例随便起的名字而已。然而，构造函数 `Object()`的名字可不能乱起，它必须得和类的名字完全一样。

我们可以把这些都告诉学生。学生可以记笔记，然后温习一遍。对于一个第一次接触对象实例化的人来说，这行看似浅显的代码看起来是这样的：

```
Object myObject = new Object();
```

对于刚上手的人来说，很难理解哪个 `Object` 是类型、哪个是构造函数、哪个是实例名称。如果把这行代码遮起来，然后让学生重写，

那么即使他们写出 `Object Object() = new myObject;` 或 `my new Object = Object();`之类的，你也不要惊讶。

让我们把这个例子改得再浅显一点。这样重写会好些：

```
Object sally = new Object();
```

这样就明白些了。`Object` 仍然很重要，但很明显它和对象实例的名称之间并没什么联系。现在也比较容易看出，实例的名称要放在类型右边。

不过，对于第一次写面向对象程序的人来说，这还是有点云里雾里。在我们能够创建的对象的所有类型里，叫做“Object”的这个很可能是最抽象的一个。我们还是坚持那个“避免不必要的抽象”的原则吧。把这个例子再改一下，让它的名字更有描述性：

```
Human sally = new Human();
```

好啦！现在 `Human`（人）很说明问题。对于在这个星球上生活的任何人（即使不是职业写程序的人）来说，`Human` 和 `sally` 之间的关系都是显而易见的。显然，`sally` 就是我们给 `Human` 的这个实例所选定的名称。

不过还是有些东西有点难以理解。新手的下一个问题可能是：如果我们说“Human Sally”，那她是一个新的人不是显而易见的吗？那构造函数到底有啥用？

不带任何参数的构造函数在编程时是很常见的。经验丰富的程序员已经习惯了那些在实例化时不接受任何参数的类。它们就是一直不带参数，除非我们有很好的理由来加上在构造时需要额外信息的构造函数。所以，传统的 `new Object()` (或 `new Human()` 或 `new List<DateTime>()`) 对我们来说很直观。

但对于新手来说，这个看起来很愚蠢。构造函数不接受参数，而

且更糟糕的是，它们在定义中除了实例化对象之外什么也不做。这一点让很多面向对象编程的初学者感到困惑。所以有时候，即使是默认的引入方法也不一定是给新手讲授概念的最好的例子。在这个例子里，我们要是给示例构造函数加上一个（或两个）参数就好多了。

```
Human sally = new Human("female", 45);
```

问问学生能不能创建一个叫做 Harry 的刚刚拿到驾照的人[①]，他很有可能想出答案：

```
Human harry = new Human("male", 16);
```

展示例子的时候，要让它浅显易懂。抛弃惯常的讲法，换成更明确的例子。去掉那些泛泛之谈、通用名称和理论，换成触手可得的明显的东西。哪怕是对象实例化这么基础的例子，我们也改了四遍才臻于清晰。

① 美国法定的驾车年龄是 16 周岁。——译者注

第36篇　为简化不妨说谎

当你教一样新东西的时候，不要认为你所说的每一句话都必须是百分百正确的。要把一个概念从一开始就讲得非常完美，既不现实，效果也不好。高级的概念本来就比较难理解，这也正是它成为高级概念的原因。它充满各种细微差别、例外和特殊情况，不那么符合一个完美的统一理论的标准。

反过来，在学一样新东西的时候，我们极度渴望的正是那套完美的客观事实。我们想要快速了解真理，而无论它们是不是真的存在，因为这些东西都会为我们学习任何学科的知识打下基础。

所以，在你成为专家之后，要先把你所在领域中那些错综复杂的细枝末节舍掉。不要讲那些“除了”和“但要是……”之类的特殊情况，因为它们*现在不那么重要*。一开始的时候，只要介绍几条规则，让你的学生能够把概念理解得差不多即可。稍微把事实扭曲一点儿来简化概念也没什么大不了的。讲课的时候，善意的谎言并不见得是件坏事。

当我们把一个复杂的主题削减成不那么完美的一套规则之后，就为新手理解该主题打下了坚实的基础。要是在人们还没完全消化一般概念之前就急着讲那些细节和例外，他们就无法系统地学习了，他们所学的内容就会像是一堆分散的碎片。一下子把碎片拼成全图可不太容易，同时消化规则和规则的例外也是十分困难。

比如，当我教别人数据库建模的基础知识的时候，我开始时的理念总是：一个好的关系型数据库应该是完全规范的，毫无例外。根据我的“张氏定律”，任何两张表都不应有冗余的数据。

在现实中，*确实*存在需要让数据库去规范化的情况。比如说，

OLAP 数据立方[1]就是一种打破了数据库规范化这一传统规则的数据库，它允许出现冗余数据。去规范化的数据库可以遍历更少的表，进行更少的关系型联结（JOIN）操作，从而让复杂搜索查询的速度大大提高。然而，新手在一开始根本就不应该关心这个问题，应该等到他完全了解了规范化的好处之后再说。要了解地基中的裂缝，他首先得透彻地了解这个地基。

所以，要是最初的理解不是百分百正确会怎么样呢？坚实的基础会给人以动力，而这个动力会让学生更快地达到高级阶段。

① 参见 http://en.wikipedia.org/wiki/OLAP_cube。

第37篇 鼓励自主思维

就像我在前一篇里面谈到的，讲授规则的时候要把它们当成坚不可摧的自然法则。这为新手提供了一个系统的出发点。要想让学生达到专家水平，到了某个时候，就需要撤掉其辅助轮。一旦基础打好后，学生就可以开始逐步偏离规则了，辅助轮、护膝、自行车头盔也都可以不用了。

德雷福斯学习模型就提倡这一点。[①]简而言之，德雷福斯模型就是一个描述学生如何学习的模型。这是1980年由一对博士兄弟（Stuart和Hubert Dreyfus）在加州大学伯克利分校做研究时提出的。[②]

当开始掌握编程等学科的时候，就不再通过分析规则来指导工作了，因为这已经变得自然而然了。我们的思维会变得更为抽象。我们会想出多种方法来实现同样的功能目标。用不着什么秘诀了，一切都凭直觉。当学生也开始凭直觉的时候，我们就知道自己教得不错了。我们要鼓励这种自主思维。

怎么才能做到这一点呢？过一阵子之后，你就会发现学生的技术问题越来越少，而策略问题开始增多。这就是他们已经掌握了“如何”和“什么”并开始琢磨“为什么”的第一个标志。当他们开始问“为什么”的时候，通常意味着他们觉得有更好的方法，而你所教的东西限制了他们快速形成的自然直觉。

要一直鼓励这种思维过程，不要一上来就把它拍死。让他们再想出一种方法，然后分析其优缺点。要是某种方法有明显的优势，就带

---

① 参见 http://en.wikipedia.org/wiki/Dreyfus_model_of_skill_acquisition。

② 要深入了解这个模型，请参阅 Andy Hunt 的书《程序员的思维修炼》（*Pragmatic Thinking and Learning*）[Hun08]。

领学生透彻地分析其欠优的情况，让他们了解在哪里可能会出问题。

接下来，他们提出的方法可能会越来越有说服力。到了某个时候，你甚至可能让他们自己来做出选择。也许有一天，你教过的学生在教你。这并不意味着你输了一阵，而是证明你真正学会了教学。

在本章中，我们剖析了教学的过程：引导编程学徒理解我们头脑中的东西。

把我们的方式教给客户也同样是很有益处的。下一章里，我们会谈一谈为什么教学能让我们和付钱的主顾之间的关系更为健康。

第 7 章

客　　户

在我们这一行，客户是我们的生命线。没有客户，我们做的东西到最后也无非是自娱自乐罢了。

但是，很多时候，我们和客户之间的工作关系不像是合作伙伴，倒像是对手。在理想的世界里，当我们挥汗如雨时，客户一边给我们揉揉肩膀，一边把香草冰淇淋送到我们嘴边，同时还不忘了帮我们擦擦嘴角。在理想的世界里，客户能够理解我们在把想法变成代码的过程中时常经受的痛苦。

但残酷的现实是，客户很少能够看到我们为了迎合不断变化的需求而经受的痛苦。客户只会考虑他们想要的那个新功能——那个在他们眼中“只需改这一点点”，可实际上却要大动干戈的功能。而利益相关方关心的只是盈亏而已。

其实这还好啦。和所有关系一样，客户和程序员之间关系的培养也是一个渐进的过程。要是双方都能理解对方注重的东西，关系就会变好。要和客户好好合作，首先要站在他们的立场看问题，然后再告诉他们，我们这边是怎么运转的。

第 38 篇　刁钻的客户无处不在

我们很容易抱怨客户太难缠，但是请记住，这种问题并不是只有我们才会遇到。事实上，和其他一些人比起来，我们要好得多了。

当建筑师设计一所房子的时候，业主只会看到那些很容易看到的东西。她看到的是那些外在的品质——花岗岩台面、硬木地板、天花线——而不是建筑师呕心沥血几个月才绘制出的设计图的精妙之处。

要是大厨做菜时多放了一勺盐，挑剔的客人会抬杠般地把菜退回去，而大厨的作品就会因此变得一文不值，只能倒进垃圾桶。我曾经见过趾高气扬的一家人，他们因为十几岁的儿子在饭里看到了一只毛茸茸的虫子后没了胃口，就把整桌菜肴退回厨房。整班厨师辛勤劳作的作品就这样被扔掉，只是因为顾客把一块露出纤维的姜当成了蟑螂。

我们都当过顾客。顾客很少会欣赏他们消费的产品里那些精致、巧妙的构思。这就是残酷又讽刺的现实。我要是雇了个管道工来解决淋浴器水压太低的问题，我想的就是赶紧修好。我不关心到底是主线路问题、限流器问题，还是喷头堵了。只要要价不高，能让我舒舒服服地洗澡就行了，谢谢。

我们给顾客或客户做软件的时候也是一样的道理。他们很可能不懂得欣赏我们编的软件有多么优雅，多么精巧。

这意味着什么呢？这意味着我们的主顾没人关心代码，至少不是直接关心代码。这还意味着，要我们改动软件的时候，他们全然不知修改这个代码是易是难，是根本不可能还是让人恼火。他们不知道那个“加上也许不错”的功能是不是“编起来会令人崩溃”。

这挺让人郁闷的，确实如此。但不要顾影自怜，因为郁闷的人不止你一个。

第 39 篇　软件黑魔法揭秘

那我们如何让客户欣赏我们的劳动呢？

有时候，要告诉客户我们是如何完成我们的工作的，尤其要告诉那些从未自己开发过软件的客户。即使在我们看来是完全显而易见的事情，对于其他人来说也并不是常识。从业以来，我已经经历了很多次这样的教训了。

许多年前，我接了一个零活，给客户做一个在线推荐系统。当时我 22 岁，还是个编程新手，那个项目听起来很简单。该软件会做一个数据库搜索，给酒吧和餐馆的酒类批发提供最低的报价。

这再简单不过了。

一个周末的下午，我和客户在一个咖啡馆碰头来讨论细节。我以为我会拿到一张井井有条的表格，里面有我要的数据——酒的品种、品牌、分销商、地址和成本。用户要查询某一种酒，点击搜索按钮，然后软件就会在系统中找出符合输入的搜索参数的最低价格。这听起来棒极了。我的客户和我都同意了这个做法，然后各自分头行动。他负责整合数据，我则开始开发这个简单、优雅的精致软件的典范。

根据我们前些天初次会谈的安排，我差不多花了一个星期把软件的基础搭好了。于是一周之后，我们一起吃午饭来看看他整理的 Excel 表格。他的表格比我想象的要复杂点儿，不是我预计的那种简单的五列表格。我咬了咬嘴唇，局促地笑了笑。

“你看，价格是随着购买量而变化的。”我的客户说。还有另外两列：针对某个特定价格，必须购买的最大和最小数量。

成吧，我想。毕竟这就是批量采购的目的所在嘛。我回到白板前，把数据模型稍稍改动了几处，行了。

第二天我就有了解决方案。我在数据库里又加了两个字段：BeginRange 和 EndRange。我把软件改了一下，让它可以接受数量，然后调整了 SQL 查询语句，这样就可以筛选出满足 @quantity>=BeginRange 且@quantity<=EndRange 的记录。这系统又变得完美了！

我们第三次碰面的时候，客户似乎有点儿困惑。我的代码很漂亮，但系统的行为似乎缺了些什么。我们试着用了用这个软件，他发现还少了几个开关。

我很快发现，在实际情况中，如果客户把相似的产品绑在一起购买可以拿到折扣。批量采购的概念并不光适用于单个产品，比如，购买了 X 件威士忌可能会让你在购买 Y 件苦艾酒时拿到折扣（有人要喝曼哈顿鸡尾酒吗？）。此外，折扣的幅度还取决于你买多少威士忌。我们可能还应该免费赠送几罐马拉斯奇诺樱桃呢。

从他的角度来看，这个系统的反应是不对的。根据他的经验，通过直接给分销商本人打电话而拿到这些折扣是很平常的一件事。但从我的角度来说，我根本没有能够推导出这些东西的数据或者条件逻辑。另外，就算他能够提供我需要的所有数据，我还得花很多时间来搞清楚如何组织这些数据。我需不需要再建些存储依赖关系的表格，来处理因购买其他产品而产生的折扣呢？我要不要搞一个“常见混合搭配”的功能，以便软件预知顾客可能购买哪些产品来获得其他折扣呢？最重要的是，在这之后还会不会出现更多让人抓狂的模型呢？

有一天，我突然明白过来，我不是在制作一个通过挖掘数据表来找出单个正确答案的简明清晰的系统，而是在制作一个名叫 Larry™ 的酒类分销经理。我在努力作出那些无法轻易推断出的决策。Larry™ 能够基于他和客户之间的关系、他四十年的从业经验，再加上点第六感来提供报价。他知道怎样让客户成为回头客，从而胜过与他竞争的

另外一百个分销商。

为啥一开始没人告诉我所有这些信息呢？我的客户是成心不告诉好奇心重的我，还是他自己当时也不知道这些很重要呢？

现在，很多非技术人员都对软件存在一个很基本的误解。作为程序员，我们基本上是逻辑和信息的整理者，我们的工作主要是推送、接收、处理和显示数据。我们中大部分人搞的都不是人工智能。我们没法轻易地写出能够推荐或猜测的程序，即使“推荐”或“猜测”是一些预定好的、可以描述的逻辑。然而有时候，在外行人的想象中我们的工作方式是这样的：有某种黑魔法盒，即使我们不给出所有的信息，它也能理清所有的头绪。

当我开始向客户解释我所开发的软件和他所要的软件之间的区别时，我的客户说他会给我答复的。多年过去了，我的代码仍然沉睡在那个被我爱称为*集尘器/镇纸石*的旧笔记本里。

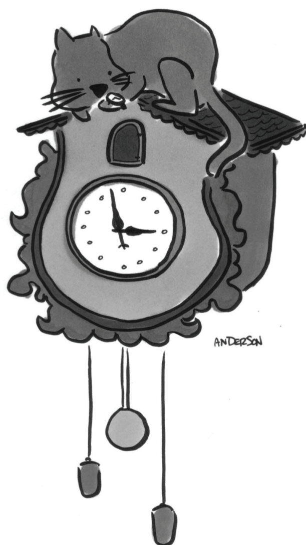
回过头来看，我更清楚地了解了客户和程序员之间的关系。在我的客户看来，Web、软件、数据库……所有这些“技术”的东西都是一团神秘的魔法云。他在潜意识中相信代码能够神奇地处理一些没有定义的逻辑，尽管这些没有交代的东西正是让此类软件编写起来无比复杂的原因。

在任何项目里都会有一些未知的东西，这就是我们的工作性质。转入开发的时候，很少会有一个想法能够完全清楚地陈述出来。而且，就算所有人都觉得它很清楚了，实际上也不是这样。在软件仅仅停留在口头上的时候，怪异的地方总能够逃过我们的思维，直到我们开始动手写的时候它才鬼鬼祟祟地揭开自己的面纱。

可能这就是我们和客户之间爱恨交织的原因。有些事情在我们这些在“黑箱内部”饱受折磨的人看来再明显不过，可外部的人却毫不知情，因为想法的真正实现都是在箱子里面进行的。我们对箱子里面

的事情了如指掌，知道里面是不是有具体的东西，而真正的痛苦和挣扎也都在箱子内部。

外部人的挫折感也来源于此。从他们的角度看，他们已经给了你很多信息啊，要求啊，细节啊——肯定足够你开工了。他们告诉我们想要什么，然后就开始等待……



接下来，几天、几周或者几个月以后，如果他们看到的东西不是他们想要的，他们也会感到泄气，因为这个箱子并没有他们想象的那么神奇。

出现这种情况的时候，把你的客户带到“箱子”里。如有需要，早点儿把他们带进来，告诉他们你们都做了些什么工作。必要的话，带他们看看实际的代码，让他们也思考一下你在编程时自然而然想到的那些问题。

第 40 篇　设定软件的目标

和客户一起工作不是件容易的事。你（程序员/设计师/救世主）和他们（有时是不可理喻的独裁者）之间的脱节永远是存在的。

但不是所有的客户都不易相处。比较好的客户，那些我们在整个咨询生涯中都不愿意丢掉的客户，那些我们希望总能有新的项目和新的想法并会随时告诉我们的客户，似乎都有一个共同的特点。

好的客户把软件置于他们自己之上。如果软件是项目中最重要的部分，那么其他的东西自然而然就理顺了。每个关于功能的决定都可以用这个简单的问题来检验："这个功能会让软件变得更好吗？"感觉和个人目的，无论是我们的还是客户的，都不能左右结果。如果不是把产品放在第一位，客户就会用其他的方式来证明他的功能要求合理：

- "……这个是我在其他网站上看到的，很酷。"
- "……因为现在是 1996 年了，所有人都在用`<blink>`和计数器！"
- "……因为现在是 2005 年了，所有人都在用 RSS。"
- "……因为现在是 2009 年了，所有人都有一个 Facebook 和 Twitter 标签。"
- "……因为那本讲可用性的书告诉我，折叠线下面的内容没人会读到！"
- "……因为我们 CEO 喜欢粉色！"

在刚刚接触客户的时候，就把软件的目标设定好，和客户一起确定最终产品要达到什么效果，然后把它记下来。这样就会让"酷"和"尖端"之类的子弹哑火。设定好目标会让你在说"不"时更有信心。

第 41 篇　激发热情，相信自己

非程序员对于编程的一大误解就是编程完全是关于算法的，但我们知道它不是这样的。编程的艺术性一点也不逊于其科学性。我们对编程工作的热情丝毫不逊于艺术家对其工作的热情。

我们需要让非程序员认识到这一点，这样，我们就可以改变和客户之间的关系。开发不只是一项服务，而我们也不只是编制软件的工匠。

让你的客户看看你工作中精巧的细节吧。不要只是草草给出几个解决对方问题的选项，而要在给出这些选项的同时，表明你对其中一个很有信心，然后解释为什么。当客户能够通过哪怕最俗套的选项看到你的热情时，他们就更可能在与你意见相左时选择相信你，并且会把你看做你所在领域的专家。

其他行业的人如何让自己的作品吸引消费者的兴趣呢？我们很容易想到“原味主厨”杰米·奥利弗（Jamie Oliver），他推广了英式烹饪、健康饮食和手工将所有食材碾在一起的做法。我们还可以谈谈音乐家杰克·怀特（Jack White），他在戴维斯·古根海姆（Davis Guggenheim）的摇滚吉他纪录片《吉他英雄》（*It Might Get Loud*）中谈到了他血染的吉他。这很有意思，因为烹饪和音乐都直接触碰你的感官，而且人们一般都愿意听名人讲自己的技艺。

不过，让我来谈谈几个不那么出名的人物吧。

卢·曼弗雷迪尼（Lou Manfredini）被人们称为“修理先生”。他是一个精力充沛的匠人，对与修理房子有关的一切都了如指掌。他每周都参加芝加哥电台节目，帮助业主处理各种各样的问题。无论是安装新的暖通空调，修理漏雨的屋顶，还是封闭露台，曼弗雷迪尼对任

何问题都能提出建议和看法。他很关心你在儿童房中使用哪种类型的油漆，还愿意帮你把地下室的害虫除净。

曼哈顿的杰弗里 · 鲁哈尔特（Jeffery Ruhalter）是第四代庖丁传人。如果屠宰不会让你反胃的话，就去看看他是如何分解一头猪[①]或是切割一块风干的牛排[②]的吧。他独特的沟通风格洋溢着热情，他提出的建议你肯定坚信不疑。你毫无疑问会觉得他的工作很有趣（除非大脊骨牛排不合你的口味）。

曼弗雷迪尼和鲁哈尔特证明了你不需要身处好莱坞就能让工作变得吸引人。他们是这些看似乏味的行当中的英雄，我们也能成为软件领域的英雄。此外，我相信编程比修漏水的水槽更有趣些。

① 参见 http://www.youtube.com/watch?v=kA7-KCBPvss。

② 参见 http://www.youtube.com/watch?v=rQiFEhsmOCk。

第 42 篇　宽容大度，和蔼可亲

充满热情的程序员比较易怒。因不成熟的想法和功能实现而与客户争辩，实在是让人郁闷。在我们写出代码之前，常常所有想法都是有关图表、功能说明、线框图的，存在于那些号称“点子王”的脑海里。可我们才是那些执行把想法变成现实这一艰巨任务的人，且不说这个工作还经常得不到欣赏。Bug 只存在于代码里，餐巾上的涂鸦里永远不会有。

多年以前，软件是让人望而生畏的命令行工具，只有电脑狂人、书呆子和怪人才会使用，而写它们的是那些更狂的狂人、更呆的呆子和更怪的怪人。那个时候，当这种孤僻的、坏脾气的、让人生厌的人也许没什么。

如今，客户就是普通人，不是另外一批搞软件的。他们使用我们的劳动产品就像使用家具一样，很自然的。使用软件的场合和不使用软件的场合之间的界限很模糊了。技术现在是主流行业了。

这就意味着我们得根据软件的目标用户调整我们的工作方式。当一个还不太了解我们工作方式的客户提出是否可以加上一个功能时，如果这样会打破已经约定好的前提，甚至颠覆整个软件架构，我们很容易就会迅速作出反击。

不要这样，要宽容。在趴在引擎盖下埋头苦干的时候，也要理解旁人的看法。如果客户的要求不切实际，就向他们好好解释，给他们一个例子，让他们看看会“放出一屋子虫子”的情况，并提供一个变通方法来解决他们的问题。

此外，要养成和他们谈话的习惯。听听他们真实的声音，也让他们听听你的声音。拿起电话打给他们，不要只发邮件。你会惊讶地发现，一个体贴的声音能给你带来很大的转机。

第 43 篇 价值远不只是工时

我们的工作对客户来说值多少钱呢？

当我们估算一个项目的价值的时候，基本上是按照行业标准，也就是仅仅按照花费的时间来计算。我们猜测一下做这个项目大致需要多少时间，再加上点不确定性的裕量，乘上每小时的价格，然后希望在交给客户之前，这个最终估价看起来还算差不离。这里面的武断之处实在让人不太满意。

工作 200 个小时就该比工作 100 个小时贵上一倍吗？所有人，从个体合同工到市值几百万美元的开发工作室，都是按照小时计费的。

这就是我们评价工作价值的合理方式吗？我们花在咨询、设计、开发、调试和测试上的时间真的能和它的价值画等号吗？我认为不能。但很不幸，这就是世界上大多数地方采用的唯一衡量标准。

在很多公司，时间记录就等于员工对业务的价值。他们美其名曰*员工利用率*。我们向客户收费的小时数越多，我们的利用率就越高。

但这一切完全忽视了所有程序员珍惜的那些时刻，那些在开发过程中突然灵光闪现、想出更好解决方案的时刻。有了天才想法，我们只用两个小时就解决了起初估计需要八个小时的问题。更棒的是，我们的解决方式比最初规划的具有更好的扩展性。太赞了！

等等，等一下。把香槟放下。现在我们得用更多的客户工作来填满那多出来的六个小时了——如果有的话，不然我们的利用率数据就该往下掉了。

在这种公司里，对程序员价值的衡量仅仅是计算解决问题时合理用掉的小时数，而不是衡量程序员真正创造的价值，因此创造性思维就得不到任何激励。所以，我们还不如就好好遵守这个规则，填满分

配给我们的每一秒钟，也用不着去琢磨更为优雅的解决方案。在这种公司里，大多数程序员都打消了他们要更高效工作的积极性，这样才能正当地提高他们的利用率。

仅仅按照时间来计算成本时，我们就把软件的价值（以及我们提供给客户的服务的价值）等同于我们开发这个软件所花的时间。

在我看来，这根本就不是同一个指标，这两个指标的最终目标截然不同。

我们工作的价值还体现在其他地方

从客户的角度来说，很多其他的指标都能体现价值。这里仅举数例：

灵活性

大多数客户——说错了——所有客户都不可能一开始就明确地知道他们想要什么。无论是功能还是字体，客户都需要在屏幕上看到之后，才能知道什么行什么不行。我们越能够在中途灵活调整软件开发的方向，就越能够提供更多的价值。这对于客户来说非常有价值，对于那些对软件这种工具不太熟悉的客户来说更是如此。

教育

我们和客户一起工作的时候，实际上是在教他们。他们会了解 Web 中的种种细节（浏览器、网络数据分析、搜索引擎优化等）以及我们作为程序员所面临的挑战（这如何能够适应数据模型呢？怎样才能创造好的用户体验呢？）。在项目完成的时候，他们会比最初时对软件拥有更为深刻的理解。这本身就是价值所在。

亲和力

如果我们打电话时和蔼可亲，沟通时口吐莲花，客户就会觉

得他们雇的人很好。这也是价值。

专业知识

在我的公司，我们同时通过工作方式和使用的产品来推进流程。我们有自己的看法。客户找到我们的时候，常常是想要我们提供答案和建议。我们对自己的看法毫不退让，这也是价值。

速度和准时

价值不也应该等同于速度吗？如果我们可以用最开始预计的一半时间来完成一个软件，为什么我们得到的报酬也要减半？

时间可以作为内部指标

既然有这么多其他的因素都会影响价值，那应该把时间摆在哪里呢？时间也是一个关键的指标，但应当用它来判断公司或个体合同工的工作是否合理，而不是衡量工作的价值。时间记录可以回答这类问题：

- 我们以小组和以个人为单位的工作量是否平衡呢？
- 我们的流程中是不是有瓶颈点需要修正？
- 在保证工作质量的前提下，我们还能接多少工作？

按产品而不是按时间收费

那么，如果不用时间的话，我们该如何给一个项目以及我们为这个项目提供的服务定价呢？要是没能把整个行业变成华尔街股票市场那样，由大家集体决定服务 X 或产品 Y 今天值多少钱，我们该如何做呢？

一种方法是把客户工作变成一组产品来报价，就像你在市场上向陌生人卖软件一样。如果你给几个客户做了类似的东西（比如一个询价管理工具或是一个全局搜索功能），就可以按照固定价格给以后的

客户报价。当你把客户工作产品化之后，就可以给自认为有价值的东西标上一个价格，日后还可以用其他价值指标来提价。

同时，把客户工作产品化还能让我们保持诚实。假设有人需要我们刚刚给另一个客户做好的某个功能，理论上，我们不需要再像第一次开发那样花掉二十个小时，也许只需要几个小时来复制一份即可。要是把时间作为唯一的标准，我们第二次收费就只能比第一次少很多了。但这毫无道理，因为这个新功能对于第二个客户的价值和对第一个客户是一样的。

如果改用灵活性、专业知识和速度作为指标，那么第二次收取同样的钱就是合理的了。我们甚至还可以改进第一次的产品，这样就可以标上更高的价格。

我们工作的价值比滴答流逝的时间要多得多。

第 44 篇　尊重你的项目经理

如果你是一个曾经和项目经理合作过的程序员，我几乎可以肯定地说，你有过因为他们而感到郁闷的时候。当你奋力苦战一个复杂功能的时候，项目经理翩然来临，居然好意思问你什么时候能够做完。当你花了无数时间来条分缕析地解决代码中一个深层次的关键问题的时候，项目经理又火急火燎地来了，催着你修改一个无关紧要的按钮标签。哦，现在项目经理又来了，问你是不是能够今天就搞定那个客户毫无预兆提出来的新功能要求。

你知道这是什么感觉。在内心深处，你总觉得项目经理没干多少事，他无非是问你什么时候、怎么样、是不是能够做某件事，而你才是那个在细节上呕心沥血、真正付出劳动来挣工资的人，对吧？

对程序员来说，要是在工作上遇到什么难处，一般是因为软件中有需要解决的困难问题。我们会全心投入几个小时，历经磨难；我们会在挫折和狂喜的情绪中跌宕起伏。对于我们程序员来说，艰苦的工作在于管理产品。我们对软件了如指掌，正如在第 40 篇中谈到的一样，软件是我们唯一应该专注的东西。

项目管理主要是人的管理

但项目经理却有不同的目标。好的开发人员是软件领域的专家，而好的项目经理是客户领域的专家。他们和电话或邮件那头的人形成了亲密的工作关系。他们知道什么时候可以推延，或者什么东西对于客户来说是真正重要的——即使我们可能不这么看。客户工作对于项目经理来说可能是一场情绪上的挣扎。

项目管理是把双刃剑

在餐馆里，要是汤太凉了、牛排太生了或者上菜太慢了，接受抱怨的人通常是服务员。要是碰上特别糟糕的晚餐，顾客一般会要求见餐厅经理，抱怨“这是他们吃过的最差的一顿饭”。厨房里那些大厨呢？一般都没他们什么事。

可是，在那些特别美妙的晚上，如果有人吃了他们一生中最好的一顿饭，他会感谢谁呢？不是服务员，也不是餐厅经理，而是大厨！大厨们甚至还时不时能出来露个脸，接受客人的鞠躬呢。

我们这个行业也是这么回事。从客户的角度来说，项目经理代表的就是你们公司。在整个设计师和程序员的团队里，只有项目经理是需要对其他所有人的行为负责的人。如果有一个程序员兢兢业业，但他邻桌的同事却有失水准，项目经理就无可避免地要独自承担把坏消息告诉客户的任务，并忍受接下来的煎熬。

但如果一个软件能够按时在预算内漂亮地发布，享受赞扬的是程序员。我们是午餐时能够从高兴的客户那里得到免费比萨和啤酒的人。

项目管理的重要性不仅限于软件本身，而且这个工作常常还吃力不讨好。所以，下次你的项目经理要求你实现客户要求的新功能时，不要马上就愤然暴起，而要找找真正驱动这个需求的原因。你也许可以推荐一个更为简单的变通方案，也可以好好地讲出道理来彻底反驳它。

你给项目经理提供的面对客户的“弹药”越多，他们就越能够做好自己的工作。

在本章中，我们看到了客户管理的微妙之处。最后，好的客户管理常常是通过我们自身的价值实现的。如果我们充满热情、令人愉快，

那么客户也能间接地体会到我们这一行的妙处。如果我们行事坦荡、和蔼可亲，可能产生冲突的时刻也会变成一个重新审视软件初始目标的机会。

在下一章里，我们还是回来谈谈那些不是与人打交道的东西吧。我们和代码之间也有同样重要的一段关系需要维护呢。

第 8 章

代　　码

到现在为止，我们已经讨论了程序员生活中的方方面面，但和我们挣钱的家伙却没有什么关系。本章我们会回到主题，回到根本，回到我们用来养家糊口的这件事上。

代码是我们所用的最基本的材料，但我们看待代码的方式和其他建造者看待他们的原材料的方式是不一样的。

和其他建造行业不同，我们的材料是取之不尽的。如今我们还可以把它们分发给任何人，分发到任何地方，距离对我们来说完全不是问题。我们可以随心所欲地复制建造的东西；我们可以在已有代码的基础上快速开发，也可以对代码层进行扩展，添枝加叶；我们不需要等待材料干透或是定型。现实的物理世界中没有什么东西是这样的。正因为这些原因，人们很容易不把代码太当回事。

本章讲的是我们和代码相处的方式，这是向我们每天所使用的原材料致敬。我相信，最好的程序员真的和代码有一段情缘。他们只有在确实必要的时候才会使用代码，只有在正确的时候才会从别处借用代码，甚至会在面对合适的挑战时用代码创建他们自己的框架。

第 45 篇　写代码是不得已而为之

当纽约一座办公楼的人们开始抱怨电梯服务越来越差时，大楼的管理方找了一家咨询公司来看看哪里出了问题。咨询公司得出的结论是等待电梯的时间太长。要解决租户的抱怨，就意味着可能要增加电梯，并实施新的电脑控制来提高电梯的效率。这种改动的花销实在太大了。

大楼人事部雇用的一个年轻的心理学家走了进来。他的建议是在电梯厅里放上镜子。等待电梯的时间长并不是问题，无聊才是问题。

他的建议奏效了。人们有事做了——照镜子，便不再抱怨等电梯的时间太长了。同样的问题，解决方式可以大不一样。

我们之中最有热情的人，是那些把大部分工作时间花在批判性和创造性思维上的人，他们常常是为了找到更简单、更“懒”的解决方案。答案并不总是埋头去写更多的代码这种明显而蛮力的方式。

有时候最好的答案是在别处发现的。下次遇到纽约办公楼的电梯问题时，问问自己下面这些问题：

- 有人以前做过这件事吗？我能不能拿现成的代码，这样就不用我自己去干这种脏活累活了？
- 这个功能对于软件的目标真的重要吗？这个功能是不是已经有了，只是需要不同的用户体验来实现？
- 有没有比我现在使用的更简单的编程方式？即使它没有完全解决问题，是不是也值得做出让步？
- 我能让这个工作自动化吗？我能不能编个软件来写这个算法，这样我以后就不用重复劳动了？

如果面对每个任务都直接进入“写代码”模式的话，我们就失去了一个认真思考为什么要写它的机会。反过来，如果我们能够批判性地思考为什么要写代码，就可以把大部分的编程时间花在真正重要的事情上。

第 46 篇 拿来主义的文化

关于我们用的这个材料有一个绝妙的讽刺。虽然写真正伟大的代码很困难，但一旦我们写好了这段精致的代码，就可以很容易地把它发往全世界，一点都不会失真或者变质。伟大的代码即使有很多程序员用过，也不会丧失其内在价值。事实上，恰恰相反。

作为一个社区，程序员一直都在帮助其他程序员。我们可以把同事做好的东西直接拿过来，以加快我们的很多流程。

开发软件就像逛超市

比如，我们可以用 Sass 来写更好维护的样式表，然后再转成 CSS。我们手头有 jQuery 和 CoffeeScript 两件利器。这两个优雅的框架以 JavaScript 为基础，却隐藏了它所有啰嗦的语法细节。需要一个 JavaScript 插件来用灯箱效果显示图片？至少有三十个这样的插件，而且都是专门为 jQuery 写的！[①]

很多年轻一代的程序员从来都不用写底层 SQL 数据库查询，因为对象关系映射（ORM）工具和代码生成框架已经帮我们完成了将关系数据库中的数据转换成对象的烦琐工作。需要 ORM 工具？有几百个开源和专有产品在等着你呢。[②]

在 Web 应用的层面，Rails 和 Django 等开发框架让我们可以开发数据库驱动的 Web 应用，同时省掉了用户界面和数据库中间的大部

① 有关三十个灯箱实现，请参见 http://www.designyourway.net/blog/resources/30-efficient-jquery-lightbox-plugins/。

② 有关多种语言的 ORM 列表，请参见 http://en.wikipedia.org/wiki/List_of_object-relational_mapping_software。

分手工劳动。我们现在可以在它们的悉心呵护下享受工作了。

任何工作，无论大小，无论是在哪个开发层次，我们几乎都可以找到别人写好的优美工具来满足我们的需要。大多数情况下，完全有理由去用这些工具。即使它们没有完美地提供我们想要的功能，即使我们需要稍微委屈一下自己的“口味”来适应它们，但和自己写一遍相比，这能省下不少时间，所以通常都是值得的。

比如，我从来不会考虑自己开发持续集成系统。Jenkins（原 Hudson）已经完美地做好了。我绝不会自己写数据库同步工具。我会高高兴兴地花上几百美元，买上一个类似 Red Gate 的工具，它会代我把合并数据库模式中可能出现的所有边缘情况都搞定。对于大多数开发工作来说，我会用那些在这个领域比我牛得多的人写的代码来搞定。

从这个意义上说，如今开发应用的感觉就有点像逛超市；也许开源运动更像是慈善超市[①]。我们可以把所有这些好工具都堆进我们的购物车，然后结账走人。到家之后，我们把这些好代码逐个拆开，自己花点儿心思把它们“缝”在一起，一个应用就诞生了。如今让软件跑起来真的非常快。

“快速编码”文化的副作用

谢天谢地，使用这些工具不会像其他行业使用“高效框架”一样在公众中造成副作用。只要我们的软件能设计出来，运行得也还不错，用户是不会注意我们在下面使用什么材料的。可不是每个行业都有这么好的运气。

① 慈善超市（Goodwill Store）起源于美国，主要业务是接收、处理、销售市民们捐赠的旧物，并将销售收入作为善款。——译者注

比方说食品行业吧。20世纪五六十年代的时候，快餐最时髦了。它是未来主义，是进步的象征。它特别能够适应活在车轮上的生活方式，让人们“边走边吃”，迅速享受还不错的食品。



“不错！就和我妈妈过去常点的一样！”

但是几十年之后，那些神秘奇妙的食品开始失宠了。这些食品的半自动化制作让肥胖症在我们的社会里流行开来。很不幸，恶劣的动物饲养条件、使用激素和杀虫剂是低价大规模食品生产的前提。由此种种，我们又回过头爱上了手工制作的真正食品。

幸运的是，我们的“快速代码”没有这种问题。换句话说，复辟到“从头一行一行写代码”似乎不太可能。高度依赖预制好的库和框架，快速让软件跑起来这种做法还是会沿用下来的。

但确实还是有些隐忧。在这些框架中，很多关键的谋划都被刻意地抽掉了。留下来的是快速的高层次方法，可以解决用其他手段解决就会很复杂的问题。但原作者的那些灵感，现在都深深地埋在了编程接口下面的某个地方。

如今，有了海量的高效平台，我们可能很快就会丧失对于底层如何工作的欣赏、兴趣和理解。

这样就危险了。

第 47 篇　代码是最好的初级程序员

代码使用起来如此之方便，所以我们很容易就忘了代码有多么伟大。

现在，让我们暂时忘掉最新的 jQuery 插件或是 Rails 的补丁吧。想想伟大的数学家卡尔・弗雷德里克・高斯，要是他在 18 世纪就有编程语言可用的话会是什么样子。

高斯可以用代码做的事情

关于高斯有一个很著名的故事。高斯还在上小学的时候，有一天，他的那位懒得出了名的老师让整个班的学生计算从 1 加到 100 得多少，希望这个题可以占掉学生们很多时间。可让这位老师头疼的是，年轻的高斯只用了几分钟就把正确答案告诉了老师：5050。

他是怎么算得这么快的呢？如果高斯当时有编程工具的话，他可能就写了这么一段代码来得到答案：

```
public int sum_range_of_positive_integers_to_100()
{
    int sum;

    for (int i = 1; i <= 100; i++)
    {
      sum += i;
    }

    return sum;
}
```

他可能很快地想了想，然后把程序重写为更为普遍的形式，以便应对老师可能让他算其他数字范围的情况，即：

```
public int sum_range_of_integers(int first, int last)
{
  if (last < first)
  {
    throw new Exception("Last must be larger than first!");
  }

  int sum;

  for (int i = first; i <= last; i++)
  {
    sum += i;
  }

  return sum;
}
```

当然，年轻的高斯当时没有这种选择。那他是如何这么快得出答案的呢？

他没有把数字一个个加起来，而是用了一种很聪明的办法。他不是把数字挨个加起来的，而是将数列的首尾两数相加，先是 1 和 100，然后是 2 和 99，3 和 98，以此类推。

$(1 + 100) + (2 + 99) + \cdots + (49 + 52) + (50 + 51)$

这样就出现了一个简单的模式。他发现这里有 50 对数字，每对数字的和都是一个奇妙的数字——101。

$101 + 101 + 101 + \cdots + 101 + 101 + 101$

于是将 100 个数相加的问题就简化成了一个简单的乘法运算：

$50 \times 101 = 5050$

高斯的方法只有人才能想得出来。他给一个乍看起来无比繁琐的问题找到了一种优雅的解法。他的探索比他大部分同学采用的繁琐的依次求和要好得多。最后，他得出了一个漂亮的小公式：

从 1 到 n 的所有整数的和 = $n \times (n+1) / 2$

那我们的代码是如何处理这个问题的呢？前面提到的代码也可以得到相同的结果，不过做法却很“暴力”——和我们告诉它的做法一样。鉴于今天的处理器如此之快，那么即使这个方法效率很差，运行我们代码的软件也肯定比高斯算得快得多。

不过，到了某个规模，高斯的方法就会胜出了。如果要高斯算 1 到 4 000 000 的和，那 `sum_range_of_integers()` 算得就慢了。

这就意味着，到了某个数值，高斯很可能能够打败代码，因为虽然这位天才知道用一个简单的公式就可以计算结果，可我们可怜的程序还是要这样做：

```
1 + 2 + 3 + … + 3 999 998 + 3 999 999 + 4 000 000
```

代码的迷人特质

高斯的故事可以给我们一些启迪：代码和人解决问题的方式有何不同。代码擅长处理繁琐的东西——基于算法和规则来解决问题——比人要强得多。它不仅是擅长，简直具有一个无可匹敌的、高效又廉价的初级程序员的所有特质。

代码不会偷懒

代码永远不会擅自决定抄个捷径或是偷工减料。本篇开头那小段代码会把 4 000 000 个整数挨个加起来，绝不会跳过一个。代码执行的精确度是无可挑剔的。

代码不会嫌烦

想象一下写这样一个小程序：

```
int x = 0;

while (x != x + 1)
```

```
{
  // 什么也不做
}
```

虽然这个任务看起来很不理智，但代码会兢兢业业地永远做这份无用功，除非运行时引擎能够察觉这个无限循环，从而强制中止程序。代码不会去分析一个任务是否重要，也不会在意自己是不是劳累过度，它只管任劳任怨地执行我们给它的指令。

代码不会遗忘

在本篇开头，我们是这么告诉引擎的：

> “无论何时我让你运行 sum_range_of_integers(1, 100)，建立一个叫做 sum 的整数。从数字 1 开始，把这个数值加到 sum 上，然后让这个数值增加 1。一直做下去，直到数字达到 100。然后给我此时 sum 的值。”

多年之后，我又翻出这段程序，重新调用这个方法，还是会得到一样的结果。代码永远不会忘记要它做的事情。那些用成千上万行代码搭建起来的软件系统也是如此。无论规模如何，在几天、几周乃至几年之后，代码还是会记得要做什么。人类做得到吗？

代码很廉价

让一个同事坐到桌前去把 1 到 4 000 000 的数字加起来，如果他要报酬的话，我们也会理解的。也许我们可以按照加法次数来计算佣金，或是按小时付费。回头看看第 8 篇“工作即福利”，对于这种高度琐碎的工作，福利是可以带来动力的。

幸运的是，我们从来不需要告诉代码有关金钱、市场经济或是度假别墅之类的事情。代码从不求回报。一旦教给代码做什么事情，我们就可以拼命压榨它了。不会有什么代码劳动法之类的麻烦事。

代码很快

代码执行的速度和我们做事的速度真是不可同日而语。它的速度是由硬件所限制的，随着时间推移，这个限制会越来越小。人类对这种速度实在是望尘莫及。

这些都意味着什么呢？想象一下有个求职网站上贴了这么一则广告：

勤奋的程序员求职！！！

教我做任何事，我都能很快学会，并且会马上告诉你我是不是需要额外的信息。我可以随时工作，工作量不限。我永远不会忘记你说的话，也不会抱怨工作毫无挑战。我特别擅长繁琐的工作。

- 地点：任何地方
- 请勿向本人推销服务或商业利益
- 报酬：0元

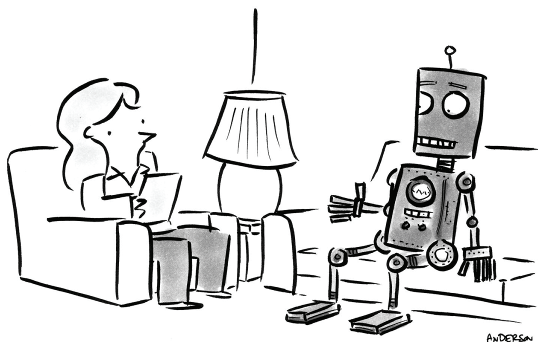
毫无疑问，代码是有史以来最伟大的初级程序员。它特别擅长繁重但能够明确定义的工作。它从不抱怨，除非我们的指令没有意义。它廉价、快速、勤勉、稳定，还不夹杂个人情感。很多公司会在一秒钟之内就雇用十名具有这些特质的程序员。

代码的威力真是可观。

第 48 篇　把机器和人的工作区分开

如果代码是初级程序员的理想“候选”，我们就应该让它马上开工。我们越早摆脱繁重的算法性工作（最适合代码做的工作）就能越早专心去做更有意思的事情。

我们编程时都有过这种经历：把一个项目的代码贴到另一个项目里，或是浪费几个小时去写我们明知以前写过的功能。有时候我们没有灵感去重新思考整个流程，结果就这么耗过一天，然后再做下面的工作。



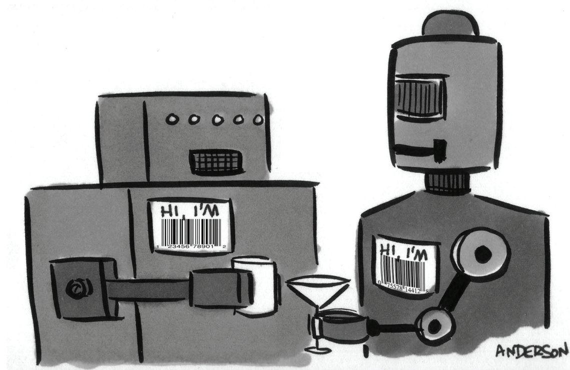
“最近我这儿看起来全都是零啊。”

这种被动的心态必须要改改了。我们没必要反复写哪怕很小的脚本，完全可以写一个程序来帮我们做这个工作。程序员的时间太宝贵了，不能浪费在重复性的劳动上。2006 年，我和伙伴一起创建 We Are Mammoth 的时候，我一直想的就是这个。

起初，我们用 C#在一个.NET 后端上制作 Flash 应用。干了几个月之后，我发现有些工作是重复性的，即我们每次都机械重复的工作。

看到同样的流程迭代几次之后，我开始把繁重却可以用算法描述的部分与每个项目中的定制部分区分开来。这种区分就像油和水一样分明。

我们做的每个应用都遵循一个公共的套路。设计好数据库之后，我们会用 SQL 写上一些存储过程，然后在 C#里面创建对象，从这些存储过程中抽取数据，变成自己的属性。然后再写上一些 Web 钩子，和用 ActionScript 写成的另一套类进行交互。只有到了这里，我们才开始在 Flash 端进行功能开发。这一大堆工作真是累人又无聊，让机器人来做比让人做强多了。



软件中有两个部件是没办法用算法生成的。首先是数据库模式。我们给汽车公司、家具厂、经纪公司、软件分销商和快餐连锁店做过软件。他们的数据库都是量身定做来解决独特的商业问题的。其次，用户界面可不是无关紧要的，得为每个客户量身定制。这些才是我们想要投入大部分时间的工作。

这时我们就停下来，花些时间来制作我们自己的代码生成器。

为了更好地描述如何从流程中把可重复的部分提取出来，想象一

下从头搭建一个博客的情景。我们从最底层的数据库开始。数据模型可能包含这样的三张表：

- Posts(ID, Title, CreateDate, Body, AuthorID)
- Authors(ID, FirstName, LastName)
- Comments(ID, Comment, Email, CreateDate, PostID)

如果你熟悉对象关系数据库建模，这个模型还是很显而易见的。文章（Posts）都有标题、正文、创建日期，并通过外键 AuthorID 来和一个作者(Author)建立关系。AuthorID 通过匹配作者(Author)表中的 ID 列来指向该表中的一条记录。评论（Comments）表则包含评论、电子邮件地址、创建日期，并通过外键 PostID 来关联原文 Post。PostID 匹配文章（Posts）表中的 ID 列。

找出可重复的编码任务

开张初期，我们开始写软件的时候，会为数据模型中的每一张表写用来插入/创建（insert/create）、读取（read）、更新（update）和删除（delete）表中记录的存储过程。这种“CRUD”方法就是在数据库中操作记录的基本过程。下面就是给 Posts 表写插入过程时要敲的代码：

```
CREATE PROCEDURE CreatePost (
  @Title NVARCHAR(255),
  @Body NTEXT,
  @CreateDate DATETIME,
  @AuthorID INT)
AS
INSERT INTO Post VALUES (
  @Title,
  @Body,
  @CreateDate,
  @AuthorID)
```

在 CreatePost 这个过程中,我们就简单地列出 Posts 表中除主键(这里是 ID 字段)之外的所有字段，然后用 SQL 写一个 INSERT 语句，加上对应的输入参数。

由于通过观察数据模型就可以精确描述如何写此类方法，用程序就可以生成任何一般性的创建方法了。前面所说的过程也可以在 Authors 和 Comments 表上重复进行。

我们可以把同样的套路用在一般性的 UPDATE、READ 和 DELETE 过程上。比如，要写一个更新过程，我们就列出表中所有字段，用 SQL 写一个 UPDATE 语句，把主键字段(这里是 ID)用作 WHERE 语句中的条件。用这种方法写出来的 UpdatePost 过程就会是这样的:

```
CREATE PROCEDURE UpdatePost (
  @ID INT,
  @Title NVARCHAR(255),
  @Body NTEXT,
  @CreateDate DATETIME,
  @AuthorID INT)
AS
UPDATE Post
SET
  Title = @Title,
  Body = @Body,
  CreateDate = @CreateDate,
  AuthorID = @AuthorID
WHERE
  ID = @ID
```

我们还能生成什么样的查询语句呢？举个例子，我们可以基于这些表之间的关系，将其推广到选择查询上。比如，文章都有作者，那我们就可以写一个 SELECT 存储过程，来查询某个作者 ID 对应的所有博客文章。我们就把它叫做 GetAllPostsByAuthorsID 吧。我们也可以写出类似的过程，来查询某个文章 ID 下的所有评论

(GetAllCommentsByPostID)。

```
CREATE PROCEDURE GetAllPostsByAuthorID(@ID INT)
AS
SELECT * FROM Posts WHERE AuthorID = @ID

CREATE PROCEDURE GetAllCommentsByPostID(@ID INT)
AS
SELECT * FROM Comments WHERE PostID = @ID
```

我们的存储过程中又出现了另外一种公式化的模式。对于表[X]中的任何外键[Y]，我们都可以写出一个这种形式的存储过程：GetAll[X]By[Y]ID。

让我们再进一步。我们可能想要按照某个字段来筛选记录。比如，我们希望获得某一天的文章：

```
GetAllPostsWhereCreateDateEquals(CreateDateParam)
```

或按照姓氏查询作者：

```
GetAllAuthorsWhereLastNameEquals(LastNameParam)
```

又出现了一个公式。对于表[X]中任何可以进行筛选的字段[Z]，给定一个参数[P]，我们就可以写出一个存储过程：GetAll[X]Where[Z]Equals([P])。

在建立 C#数据访问层、Flash 层能够调用的 API 以及 ActionScript 层本身的时候，也可以找到类似繁琐但又可以用算法描述的过程。所有这些底层的脏活累活都特别适合我们上一篇文章中的虚构求职者。或者，更现实地说，我们可以写出聪明的程序来帮我们干活。

我们每次开始新项目的时候，这些工作都不算少，但是只要找出这类工作的公式，我们就可以把这种工作交给……机器人去做。

第 49 篇　从核心开始生成代码

代码生成是每个程序员都应当经历的旅程。这可以解放思想的束缚，让我们把代码看做转变工作方式的有力工具，而不仅仅是用来写程序的材料。

那么，我们如何着手去写一个代码生成器呢？要想真正了解的话，我强烈推荐 Jack Herrington 的杰作 *Code Generation in Action* [Her03]。该书详述了生成各类代码的技巧和高层次模式。但我们开始的时候并不需要如此精妙的细节。下面就是一些基本的东西。

定义输入源

首先要创建一个输入源。我们的代码生成器所需要的所有参数都汇集到这里。输入源可以很简单（比如普通的 XML 或 JSON 文件），也可以很健壮（比如数据库本身）。

我们公司部署的第一个代码生成器 X2O 是用 XML 文件作为输入源的。XML 文件定义了我们要生成代码的数据库中的表、字段和外键。下面就是一个将第 48 篇中的博客数据模型转换成 XML 输入源的例子。

```
<input_source>
  <table name="Posts">
    <field name="ID" type="int" identity="true" />
    <field name="Title" type="NVarChar" length="100"/>
    <field name="CreateDate" type="DateTime" />
    <field name="Body" type="NText" />
    <foreignkey name="AuthorID" to_table="Authors" />
  </table>
  <table name="Authors">
    <field name="ID" type="int" identity="true" />
```

```
    <field name="FirstName" type="NVarChar" length="50"/>
    <field name="LastName" type="NVarChar" length="50" />
  </table>
  <table name="Comments">
    <field name="ID" type="int" identity="true" />
    <field name="Comment" type="NText" />
    <field name="Email" type="NVarChar" length="100" />
    <field name="CreateDate" type="DateTime" />
    <foreignkey name="PostID" to_table="Posts" />
  </table>
</input_source>
```

随着时间的推移，你的输入源也会随之增长。你会发现越来越多的东西要生成，也就需要更多类型的输入。比如，在 X2O 的第一个版本开发完成几个月后，我们想要给代码生成器加上创建文档的功能。我们给每个 `table` 和 `field` 节点都加上了一个叫做 `friendly_description` 的属性，这样就可以参考这些属性来为 ActionScript 代码生成 API 参考文档了。

选择合适的编程语言

应该选择适合做代码生成的语言来编程。用来写代码生成器的语言并不一定就是生成代码的语言。在 X2O 里，我们用 C#来写代码生成器，但是输出则有 SQL、C#、HTML 和 ActionScript。

所选的语言必须具有输入输出功能，这样你才能把生成的代码保存在机器上。幸运的是，如今几乎所有的主流编程语言（C、C++、C#、VB、Java、PHP、Python、Ruby、Perl）都支持这个。如果你从来没写过读写文件的程序，就花上一个小时来研究一下吧。这个工作你的代码生成器要做上很多。

Herrington 优选的语言是 Ruby，因为它对输入输出和文本模板工具（如 ERb 和 ERuby）的支持很好，对于他举例的输入源语言——

XML 的操作也很方便。

在输入源中提取有用信息

有了输入源，写一个程序来从中提取有用的信息吧。在上例中，我们把 XML 文件的内容映射成其 C#中的对象。这样，你既有了方便构建输入源的系统（XML），又有了根据输入源生成代码后方便操作的系统（比如 C#中的对象）。

在今天的体系中，E4X（ECMAScript for XML）等语言已经让将输入源转化成可编程对象的工作变得非常顺畅了。无论你用什么方法，能够方便地浏览和审查输入源是非常重要的。你在下一步中就知道为什么了。

给你的输入源提供者配上模板

有了可以用的编程环境和输入源，下一步就是要写模板。在我们的博客例子中，开发过程中每个繁琐的部分都有一个公式。比如，为了生成所有的 CRUD 语句，我们无非是遍历数据模型中的每张表，然后对每个表应用相同的语句。以 SQL 的 CREATE 语句为例，我们可以先拿来下面的真实 SQL 代码：

```
CREATE PROCEDURE CreatePost (
  @Title NVARCHAR(255),
  @CreateDate DATETIME,
  @Body NTEXT,
  @AuthorID INT)
AS
INSERT INTO Post VALUES (
  @Title,
  @CreateDate,
  @Body,
  @AuthorID)
```

然后把定制部分换成可以替换的变量：

```
CREATE PROCEDURE Create[cur_table] (
[List_of_attributes_as_input_params])
AS
INSERT INTO [cur_table] VALUES (
[List_of_attributes_as_SQL_insert_params])
```

这样就有了生成 CREATE 语句的模板。

有了这个模板，我们就可以遍历输入源提供者中的每个表结点，填上相应的值。在我们的例子里，cur_table 就是每张表的名称，而 List_of_attributes_as_input_params 和 List_of_attributes_as_SQL_insert_params 则是从字段结点中得出的。

用伪代码描述的话，创建生成代码的过程就是这样的：

(1) 为希望生成的代码制作一个样例文件。

(2) 抽掉定制部分，换成变量，这样就得到了模板。

(3) 编制读取模板的代码，遍历输入源，根据需要替换模板文件中的变量。

(4) 把新生成的文件保存到磁盘上。

(5) 最后对文件进行某种操作（运行、编译等）。

组件驱动式设计

有一条经验是把所有的生成器都变成独立的库。X2O 在早期的时候就是一个有一大堆代码的大文件。生成数据库、SQL 脚本、数据访问层、Web 服务、Flash 对象和 CMS 文件的代码全都堆在一个库里面。虽然这也能工作，但随着增长它变得没法管理了。维护起来越来越难，因为对生成器的任何一点小小的改动都意味着要重新编译上万行代码。

后来我们把各个部分拆开，得到了大概三十多个独立的库。这下

维护起来容易多了。我们还有一个统揽全局的主生成器库，通过引用就可以把所有的生成器连接起来。如果不是一直需要某些生成器的话，还可以把它们关掉。

不管怎么说，封装和组件化都是很好的编程习惯。但如果我们要建立好几十个小的生成器的话，这两点就格外重要了。

有了这五个简单的小窍门，我们就可以开始了。

自动化时要小心

那代码生成有什么不好的地方吗？是不是有些时候我们不应该用呢？是的。下面就是你在自动化过程早期可能会常犯的错误。

避免手动调整生成的代码

制定一条严格的规则，就是代码在生成之后是不能修改的。生成的代码就像精美的瓷器一样：你打碎，你赔钱！

如果你生成代码只是为了后续再去鼓捣鼓捣，那么这可能让我们的流程更累人，一点都不会轻松。为什么呢？比方说我们给数据库加上一个字段，希望按照更新后的数据模型重新生成代码。每次生成的时候，我们都得记住曾经手动修改的东西，还要保证代码会原样重新修改一遍。

如果生成的代码确实需要调整，那也有优雅的解决方法。在 C#里，我们可以把类标为“部分类”（partial），这样我们就可以在多个源文件里定义一个类。在 X2O 里，每个生成的 C#类都是部分类，这样就算有需要，我们也可以在另外的文件里对同一个部分类添加新的方法或属性。

要是在你用的语言里没有部分类可用，也仍然有其他优雅的手段。比如，你可以扩展（extend）类或是编写定制的辅助类（helper class）。

生成的代码要和真正的代码一样整洁

手动编程的时候，因为我们希望以后维护起来容易，所以保持代码整洁特别重要。使用代码生成器的时候，我们从来不需要去实际维护生成的代码，只会维护生成器，结果这造成我们没有动力让生成的代码也同样整洁。

也正是因为这个原因，有些人认为代码生成是在快速输出和定制代码之间做出的取舍——它生成了许多最终应用程序很少会用到的冗余代码。因为从程序里冒出一堆堆的代码太容易了，所以我们可能不那么在意它生成的代码是不是简洁、优化。但这个问题实际上很好解决。

我们的下一个项目可能不需要某一套数据访问方法。我们可以用输入源来定义一些选项参数，这样就不会生成一大堆项目不需要的代码。随着生成器越来越成熟，我们可能会关掉某些选项，不去生成某些代码。这也是组件式设计真正起作用的地方。

有些人说代码生成器生成出来的代码不够优雅，但这和代码生成器没什么关系，它只和我们让代码生成器生产什么有关。

如果生成的类里有重复的函数或公共方法，我们可以重构代码生成器中的模板，还可以把重复的函数写成独立类，放在生成器之外。我们还可以把手动编程中的技巧应用到生成的代码里。

对于代码生成来说，并没有什么东西阻止我们遵循好的编程原则。

知道不要生成什么

虽然代码生成让你更辩证地思索日常工作中的模式，但不要强加这些模式也是很重要的。在代码生成取得了几次甜美的胜利之后，我们可能会觉得所向披靡，想把所有东西都放到代码生成器里面，即使是那些无法自动化（但肯定很累人）的东西。我们很可能把太多需要定制的东西也放到自动化的机器里了。

这时我们就需要好好考虑代码生成的好处了。如果输出代码需要太多的定制输入，或用起来需要调整的地方太多，那么也许我们一开始就不该去生成这部分代码。就像坏代码有一股不好的“味道”一样，不好的代码生成也有一股难闻的“味道”。

写代码生成器可以让我们仔细考虑什么样的工作繁琐但可以自动化，什么样的工作仅仅是繁琐而已。

第 50 篇　自主开发的情形

代码生成和对象关系映射器并不是什么新鲜玩意。我们在 2006 年制作自己的公司框架的时候，市面上已经有很多类似的东西，我们也可以直接拿来用。当然，要是这样的话，开始的时候就可以给我们省下无数的时间了。从零开始制作任何框架都是一项让人望而生畏的工作，特别是我们知道其他框架已经开发好多年了。

于是，在如今这个拿来主义的文化背景下，自然而然地就会产生疑问：如果已经有了可能毫不逊色的产品，为什么你非要开发自己的框架、平台或插件呢？

对于我来说，有三大原因。

深刻理解问题空间

写自己的工具的时候，我们别无选择，必须把自己完全沉浸在这个工具极度渴望解决的问题里。我们必须成为这个领域的专家。“嗯，我下载了这个库，复制了这段范例程序，改了改参数，然后……我也不知道，不过它似乎就跑通了。”这种事是行不通的。没有多少事比一个程序员只知道程序能工作却不知道为什么更让人不安的了。

比如，很多对象关系映射（ORM）对数据都是默认延迟加载的，只在对象中的关系数据被访问的时候，才从数据库里读取这个数据，而不是一上来就从数据库里读入关系数据。这一点特别高效，因为框架只有在我们请求数据的时候才会做数据库查询。在任何时候，存储在内存中的数据都会相对少一些。

对于新手 ORM 用户来说，这种知识只是可有可无而已。他可能只是对其编程，却不怎么关心这个概念。在他的测试环境里，数据库

只有稀稀拉拉的几条测试记录，他写的利用 ORM 取回数据的代码运行得漂漂亮亮。就这么成了。

但一旦他把代码放到线上环境中，真实的数据开始涌入的时候，他写的这段看起来无害的调用 ORM 的小代码一下子就让服务器瘫痪了。比如，这段貌似无辜的代码：

```
foreach (Person person in myCompany.RelatedPeople)
{
  s.AppendLine(person.RelatedOffice.City);
}
```

是在对公司中的每个人都进行一次数据库查询，获得他们的办公室地址。这几行代码就可能意味着几千次查询。

如果我们是开发工具的人，我们肯定要知道所有可能捅娄子的地方。如果我们只是用其他人的框架，就可能无法觉察这些问题，等发现的时候已经太晚了。

发现核心问题并加以改进

不管那些汗牛充栋的预制工具中都有些什么，你都可以找到某个小东西加以改进来适应你编程的方式——更好地适合你开发的软件类型。

就我自己的情况而言，理由很简单。没有一个框架能够定制生成直接连接数据库模型的 ActionScript 对象。如果我用现成的东西的话，很可能得用两套软件：一套来建立我的.NET 层，另一套用来创建 ActionScript 类。如果我改动了数据模型（这经常发生），我就得分别重建软件中两个不相干的部件。

X2O 在很多方面都和其他代码生成器差不多，可以快速生成映射到数据库模式的对象，但它是专为我们业务的核心需要设计的，没有

任何一种工具和 X2O 生成代码的方式一模一样。随着最初几年的业务增长，我们根据开发流程来量身定做软件，而不是让我们无法控制的软件决定我们如何工作。专为数据库驱动的 Flash 应用制作一个代码生成器是我们想要与现有产品进行的一场战斗。这是一个我们想要更好解决的问题。

Stack Overflow 的开发团队也走过了同样的道路，编写了自己的微型 ORM。当他们的编程论坛网站最初发布的时候，他们选择了 LINQ-2-SQL 来处理所有的数据库查询。但随着流量和存储数据量增长到一定程度的时候，性能损失就很大了。

鉴于他们支持的流量以及使用 ORM 的方式，LINQ-2-SQL 解释 LINQ 查询，生成相应的 SQL 查询，然后执行查询的方式实在是效率太低了。传统 ORM 上查询的执行速度太慢了。

他们的资深工程师 Sam Saffron 和 Marc Gravell 没有把它换成另一个现成的 ORM，而是开发了自己的轻量级版本，叫做 Dapper。[①]Dapper 的执行速度要快得多，因为它没有 LINQ-2-SQL 将查询转化成关系对象时的很多额外开销。它只接受原生 SQL 输入，绕过了将一种特定域语言（如 LINQ）转换成另一种（如 SQL）时与生俱来的一些瓶颈。

不过，你虽然提高了性能，却损失了稳健性。Dapper 和大多数传统 ORM 不同，它不会自动将查询映射成对象的关系。

在 Stack Overflow 的代码里，最初 Dapper 只是在有性能瓶颈的地方使用。在这些地方，损失一些对传统 ORM 编程时的优雅来换取执行速度的提升是值得的。如今，他们几乎所有新的工作都是用 Dapper 完成的。通过自主开发，Saffron 和 Gravell 揭示了瓶颈所在，并且更

---

① 他们写 Dapper 的原因请参见：http://samsaffron.com/archive/2011/03/30/How+I+learned+to+stop+worrying+and+write+my+own+ORM。

好地解决了自己的特殊问题。

程序员的傲骨

开发自己的工具集来完成工作是对我们劳动的最高奖赏。对于程序员来说，有什么比开发出能够让未来的编程变得大为容易的工具更有成就感的呢？开发自己的东西总是让人干劲十足。

开发自己的工具集的时候，我们对于自己工作中珍视的东西有了更多的了解。很多时候，这些工具就是决定职业生涯的东西。这是因为它们不仅可能会帮到其他程序员，也能揭示我们决意改进的东西，还让我们坚定的立场浮出水面。

我们开发的工具也是自己的。它们充满了我们自己关于某些事情应当如何完成的独特见解。没有人能够告诉我们应当如何开发帮我们自己做事的工具，所以在写这些小工具的时候，我们能够决定什么是重要的。对于某些人来说，为自己开发工具，就暂时不用为他人写代码了。这就是我们作为程序员的自豪感闪耀的地方。

自豪感，我们的旅程也就以它收尾吧。

第 9 章

自 豪 感

有一天，我在《纽约时报》的社论版上看到一篇标题为“建筑工作的治愈力”（The Healing Power of Construction Work）的文章。[1]文章中，一位来自中美洲的木匠谈到，他雇用的建筑工人中，多数都和他一样，曾经触犯过法律。他最好的一些工匠吸毒成瘾，曾犯重罪。里面甚至还有个假释的杀人犯。

他想说的并不是建筑工作会吸引暴力的人，相反，它为这些命途多舛的人提供了一个“疗伤”的场所，让他们可以逃避过往。

我们做手工工作的时候会很平静，利用原料制作东西的时候，精神也会很专注。这位木匠的雇工不是把建筑仅仅当做一份工作，而且还是对现实的一种逃避，是一个真正把事情做好的机会。

建筑工作本身会带来一种回报：创造之前并不存在的东西所带来的满足感。只要愿意学习，努力工作，关心产品，建筑工作是任何身强体健的人都能够做的事情。在这里可以获得成功，即使是那些在生活的其他方面未曾体会过成功的人也有机会。

这篇文章让我深感触动，因为我看待编程的方式正是如此。我没有犯过重罪，也不认识哪个程序员是在逃犯。不过，我知道很多人，

① 参见 http://www.nytimes.com/2010/08/22/jobs/22pre.html。

不管他们承认不承认，都相信编程有一种逃避现实的抚慰作用。编程给你带来了从无到有的创造的快乐。

我认识的大多数程序员甚至不关心开发的是什么或是为谁开发。只要是在解决一个有趣的问题，只要有机会能够优雅地创造一些东西，他们就感到满意。将一个问题条分缕析，然后解决得巧夺天工，这种精神鸦片让程序员不能自拔。

我们之所以开发和设计软件，是因为我们真的热爱做这个，无论这种感情是溢于言表还是藏在内心深处。我认识的最优秀的程序员会不断雕琢每一个开发决定，有时甚至是无关紧要的开发决定。正如那些建筑工人一样，我们并不仅仅是写代码，而是要把代码写好。

对热爱这份工作的人来说，工作并不仅仅是为了挣钱。更容易的挣钱方法也有。这个职业完全是我们自己的选择。

9.1 形象是个问题

问题何在？我们这个挺小的圈子之外的人，很少能够认识到软件开发能够带来多少回报，甚至我们当中很多人也没有完全认识到这一点。这也就是我自称 Web 软件应用程序员时浑身不舒服的原因。这种说法里面透着那种随遇而安的世故。也许这主要是我们工作的性质决定的吧。

在人生的低谷，我们情绪低落，满面愁容，不停地跳槽。我们的困境和其他行业并没有什么不同，但我们感到耻辱是因为此时和我们最有激情时的状态有巨大的落差。

我们完全投入工作的时候，会比大多数人更沉浸在自己的思绪里。双手打字，双眼紧盯着屏幕出神。望着窗外，若有所思，可实际上我们什么都没看见，只有伪代码在头脑中运行。面无笑容，并不言语，也不期待有人答话。我们此时只愿独处，与自己的思绪独处，无论世事变幻。这就是富有激情的程序员完全从周围的世界中逃脱出来的情景。

但当我们毫无生趣的时候，我们也很安静。面无笑容，不想和人说话。唯一的区别是，我们打字时有气无力，眼望窗外时注意到了周围的世界，并且想要走出门去。如果有人鼓励一下，我们会长叹一声，捶击桌面，再咕哝几句我们如何不满于手头的工作。闷闷不乐的程序员可能和心满意足的程序员看起来差不多，差别不过是一声长叹而已。

所以，我们有了*形象问题*。世界上其他人都把程序员看成一帮戴着耳机的隐居型怪人，可我们实际上是富有激情的手艺人和思想者。咋会这样了呢？

9.2　烹饪行业的一课

比方说烹饪行业吧。Emeril Lagasse、Bobby Flay、Mario Batali 和 Gordon Ramsay 都是热情洋溢（有时候甚至有点烦人）的大厨，热情简直要从他们的毛孔中冒出来。他们的热情不仅感染了其他厨师，更感染了大众。我们（不那么出名）的同辈就没有这种国际魅力了。程序员中没有什么名人能够感染程序员之外的人群。

乍一想，你可能会觉得这是因为相对于编程来说，人们一般更愿意烹饪。不过，我可以向你保证，虽然我每次在大厨制作“碎辣根三文鱼配焖蔬菜和土豆泥”时都会垂涎三尺，但我可不会去自己做这道菜。烹饪行业已经找到了一种将手艺推销给所有人的办法，即使我们中很多人从来都没有摸过锅。

也许这就是因为我们爱吃嘛。食物看起来就诱人。观赏某人烹调的过程激起了我们最原始的情感。有些人把它称为“食品色情”（food porn）。但如今的食品热潮并不是由来已久，烹饪节目毕竟已经存在几十年了。我们大多数人都听说过 Julia Child，但你听说过 Justin Wilson、Jeff Smith 或 Graham Kerr 吗？他们的烹调节目也上映多年，但当时的社会对烹调没有这么大的热情，他们从来没有像如今的同行那样获得的广泛认同。那么是什么东西发生变化了呢？

过去，烹饪节目看起来就像在奶奶的厨房里：几部摄像机，一个老学究在那念叨要加上小半杯这个，一茶匙那个。烹饪节目是给那些愿意烹饪的人拍的，它们从来没有影响到目标观众之外的人。烹饪无非是烹饪。如今，节目的切入点可是完全不同了。

首先，它们强调细节。这里有一个特写，然后又有一个大特写——你可以看到厚厚的牛里脊肉上的花纹，还可以瞄一眼主厨的指甲是不是干

净。高清电视帮助了食品行业，当然也帮助了其他行业来进行销售。过去，牛排就是牛排。如今，是关于肥瘦相间的精美花纹、流淌的肉汁和烤架印。细节正是吸引力之所在。

其次，如今的节目让烹饪变得平易近人。电视烹饪节目只是照着菜谱做菜的日子已经过去了，如今的节目强调“简单”。每个人都能做。三十分钟做好一顿饭，五美元做一道菜，然后就可以和朋友们享受美好时光了。烹饪变得简便易行而富有乐趣了。

大厨把他们手中的食物描绘得如御用珍馐一样。他们对于原料和口味的描述都充满激情，即使只是用那些平淡无奇的形容词：新鲜、可口、美味。如今，大厨总是最先品尝自己的作品（一般是在节目的高潮结尾），用非常夸张的“嗯嗯~~~”来赞扬菜肴对味蕾所施的魔法。

更有甚者，坏的东西也能大卖。找个礼拜五晚上，到一个真正的餐厅厨房去看看实情吧。大喊大叫、汗如雨下、食物掉在地上，简直是危机四伏。在 Anthony Bourdain 的《肮脏的点滴》（*The Nasty Bits*）一书中，“系统 D”描绘的景象和电视上展现的一尘不染的烹饪世界可谓大相径庭，这本书可是跻身《纽约时报》畅销书排行榜的哟。

在电视上，Gordon Ramsay 让许多濒临倒闭的餐馆中的情形变得臭名远扬。《厨房噩梦》（*Kitchen Nightmares*）节目展现的是餐馆经营得糟糕透顶却依然营业的事实。看着一家餐馆从几乎无可避免的灾难中脱身而出，显然这挺有意思的。烹饪行业已经学会向大众推销自己的商品了。

其他行业的领跑者也已经找到了自己的灵丹妙药。他们呈现自己手艺的方式能够触动我们的神经，让本来没有什么兴趣的人也开始关心。

不同意这个说法？随便哪一天晚上翻翻各个电视频道吧。过去几年里，在美国和欧洲，涌现出了很多大热的节目：有关抓螃蟹的（《致

命捕捞》，*Deadliest Catch*）、训犬的（《狗语者》，*The Dog Whisperer*）、少年合唱团的（《合唱团》，*The choir*）、减肥的（《超级减肥王》，*The Biggest Loser*）、带孩子的（《超级保姆》，*Supernanny*）、蓝领脏活的（《干尽苦差事》，*Dirty Jobs*）以及抚养八胞胎的（《十口之家》，*Jon and Kate Plus 8*）。这些行业本来都算不上特别光鲜的。



"一个姑娘，一个魁梧的单身汉，还有七个小矮人？！
你确定这不是真人秀？！"

那为啥软件开发就不行呢？我们为什么就不能像别人一样，对自己的行业进行营销宣传呢？代码可以让我们玩游戏，交朋友，在这个世界上任何一个角落和他们聊天，找到真爱，购买东西，监控病人，打理生活，还可以做其他许许多多的事情。我们每天都在创造这些神奇的工具，我们要讲的故事引人入胜。只要问问那边那个一言不发的家伙就行了。

我并不是在建议近期开始试制《顶级程序员》或是《编码梦魇》之类的节目，但我们应当让自己多在公众面前曝光。我们生产了如今社会赖以运行的工具，我们中的一些人每天都在努力让它们变得比过去更快、更廉价、更漂亮。编程是一项让人着迷的工作，而把它展现给其他人是我们的责任。

软件世界在好的时候就像一个井井有条的厨房，坏的时候就是彻底的组织噩梦。传统在其他领域通常意味着宏伟和永恒，但在软件行业则不同。我们所处的环境也在不断变化。如今我们使用的东西在五年后看起来就会像古董一样。这些都是可以讲给公众的话题。

做这件事需要循序渐进。第一步就是我们每个人对待工作的方式。在餐馆里，好的侍者为展现一道菜而自豪。这不光是大厨的一道菜，也是侍者的一道菜。手艺就是精美的佳肴与单纯的食料之间的区别。从这个意义上讲，我们也应当为自己的工作感到自豪。我们应当经常摘下耳机，尽可能多地和非技术人员交流。软件开发也有很多故事可讲。

开发软件的过程同样可以是饶有趣味、引人入胜的。我们做的确实是一个可以营销的行业。这要靠我们来让它变得不局限于代码，正如烹饪行业已经超出了配料的范畴。让我们一起用热情把它散播给其他人吧。

这是我每天都要经历的挣扎。每当别人问起我是做什么的，我都耸耸肩。

我想说，我是一个 Web 程序员和设计师——如果你愿意，可以说是“一个现代程序员”。但“程序员”并不能引起我想要的那种反响，它缺乏医生、建筑师或美国总统的那种气概。“医生”意味着创造奇迹、妙手回春，“建筑师”暗指梦想家和建造大师，我听说当美国总统也是有些福利的。

对外行来说，程序员就等于“搞电脑”，这也不比把外科手术等价于“摆弄锋利的东西”好到哪里去。下次有人问我是做什么的，我就说我是乡村歌手好了，起码这还容易点儿。

事实上，有时候我们是集医生、建筑师和统治者于一身。我们用代码创造奇迹，让梦想驰骋，苦心建造，然后指点江山。他们要是问我是做什么的，我就给他们看看这本书。

参考文献

[CB06] Ka Wai Cheung and Craig Bryant. *Flash Application Design Solutions: The Flash Usability Handbook*. Apress, New York City, NY, 2006.

[FH10] Jason Fried and David Heinemeier Hansson. *Rework*. Crown Business, New York, NY, 2010.

[Gol05] Natalie Goldberg. *Writing Down the Bones: Freeing the Writer Within*. Shambhala Publications, Boston, MA, 2005.

[HH07] Chip Heath and Dan Heath. *Made to Stick: Why Some Ideas Survive and Others Die*. Random House, New York, NY, USA, 2007.

[Her03] Jack D. Herrington. *Code Generation in Action*. Manning Publications Co., Greenwich, CT, 2003.

[Hun08] Andrew Hunt. *Pragmatic Thinking and Learning: Refactor Your Wetware*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2008.

[Ker04] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, Reading, MA, 2004.

[Pin09] Daniel H. Pink. *Drive: The Surprising Truth About What Motivates Us*. Riverhead Books, New York, NY, USA, 2009.

“这是Pragmatic Programmers系列中的又一本书——对新手来说是指引，对专家来说是重温，这是关于程序员技艺（和生活）的一本美妙的智慧集。”

——Derek Sivers，CD Baby和sivers.org的创始人

“Ka Wai Cheung先生为那些寻找自己赖以生存的代码的专业开发人员写了一本书。这本书不是用那些在任何博客中都能找到的传统想法拼凑起来的，而是用强有力且有针对性的方法，讲述专业编程的技艺和现实。

如果你想找一本新瓶装旧酒的编程规则，那就不用看这本书了。但是，如果你正在寻找一种视角，看看软件开发是什么，或者你想要一套由真实经验提炼出的指导方针，那这本书正是你需要的。”

——Bob Walsh，作家、47 Hats的创始人

“充满‘美味’的经验，每篇的大小也十分‘适口’——在这本书里你可以学到很多。花上些时间从过来人那里学学吧。”

——Adam Hoffman，高级开发主管

“一本好书，有现代程序员从日新月异的世界中得到的提示、技巧和经验教训。从事开发或与开发人员合作的人士不可不看。”

——Caspar Dunant，Webfish

The Pragmatic Programmers

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈／投稿／推荐信箱：contact@turingbook.com
热线：(010) 51095186转604

分类建议 计算机／程序设计

人民邮电出版社网址 www.ptpress.com.cn



ISBN 978-7-115-29508-8

定价：29.00 元